Karlsruhe Institute of Technology

Performance Optimization Strategies for Transactional Memory Applications

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Martin Otto Schindewolf

aus Eschwege

Tag der mündlichen Prüfung:	19. April 2013
Erster Gutachter:	Prof. Dr. Wolfgang Karl
Zweiter Gutachter:	Prof. Dr. Albert Cohen

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

Karlsruhe, den 4. März 2013

Martin Schindewolf

Abstract

Transactional Memory (TM) has been proposed as an architectural extension to enable lock-free data structures. With the ubiquity of multi-core systems, the idea of TM gains new momentum. The motivation for the invention of TM was to simplify the synchronization of parallel threads in a shared memory system. TM features optimistic concurrency as opposed to the pessimistic concurrency with traditional locking. This optimistic approach lets two transactions execute in parallel and assumes that there is no data race. In case of a data race, e.g., both transactions write to the same address, this conflict must be detected and resolved. Therefore a TM run time system monitors shared memory accesses inside transactions. These TM systems can be implemented in software (STM), hardware (HTM) or as a hybrid combination of both. Most of the research in TM focuses on language extensions, compiler support, and the optimization of algorithmic details of TM systems. Each of the resulting TM systems has a different performance characteristic and comes with a number of parameters that can be tuned. This optimization space overwhelms the application developer who has been the target audience for the invention of TM. In contrast to other research activities, this thesis proposes TM tools that are candidates to bridge the gap between the application developer and the designer of the TM system. These tools for TM provide insights in the run time behavior of the TM application and guide the application developer to optimize the TM application.

In contrast to related work presenting tools for a specific TM system coupled with a programming language, this thesis presents solutions that cover multiple TM systems (Software, Hardware, and hybrid TM) and account for different parameter settings in each of the TM systems. Moreover, the proposed methods and strategies use information of all different layers of the TM software stack. Therefore, static information, information about the run time behavior, and expert-level knowledge need to be extracted to develop these new methods and strategies for the optimization of TM applications. Extracting and using this information poses the following challenges that need to be addressed.

The first challenge is to capture the genuine TM application's behavior at run time. This is especially important because transactions are sensitive to artificially introduced delays of a thread which may cause an introduced TM conflict. This may lead to a recorded application's behavior that is biased through the tracing machinery. Therefore, a lightweight trace generation scheme with a small probe-effect needs to be developed and evaluated. This thesis presents similar solutions for STM and hybrid TM that generate event traces. For STM, the logging of frequently occurring events, such as TM loads and stores, quickly saturates the write bandwidth of the hard disk. Thus, a reduction of the amount of data to be written to hard disk needs to be achieved. Therefore, we research how these events can be compressed online without disturbing the application. A multi-threaded trace compression scheme is designed, implemented, and evaluated. We enhance the TinySTM, a word-based open source STM, with tracing facilities. Due to the lightweight and compressing tracing

scheme, the resulting TracingTinySTM generates event traces that capture the genuine TM application's behavior. For an FPGA-based hybrid TM system, called TMbox, a low-overhead tracing solution is shown. Each processing element is enhanced with an event generation unit for monitoring. When an instruction of interest is encountered, a corresponding time-stamped event is generated and passed on to the log unit. This unit uses idle times on the bus to transfer it to the memory of the host machine. Although all methods are tailored to and applied in the context of TM, they are universally applicable in a different context. Example application areas are the monitoring of arbitrary events in an FPGA-based processor prototype or the logging and compressing of events in a math library.

The second challenge arises with the release of IBM's new Blue Gene/Q architecture with HTM support. We establish a set of best practices for the new architecture so that the application developer can exploit the full potential of the architecture including the TM subsystem. For this purpose, we introduce a new benchmark, CLOMP-TM, that has a series of parameters that allow us to explore varying transaction granularities and conflict rates, coupled with typical computational kernels found in scientific codes. This enables new insights through evaluating the TM system and comparing the performance with other synchronization primitives of OpenMP. Further, this new BG/Q architecture also requires tool support to enable optimizations of scientific applications. A trace-based solution to capture the run time behavior on a per event basis is not feasible for this proprietary HTM system. This HTM system requires a complete software stack, including compiler and run time system. Thus, we design, implement and evaluate three different tools for TM that enable programmers to explore the subtleties of TM execution and contribute the following to the state-of-the-art:

- the first tool that profiles applications using MPI and OpenMP with TM on BG/Q,
- a tracing tool for TM that enables in-depth inspection of thread-level execution and utilization of the architecture through visualization with the state-of-the-art visualization tool Vampir,
- a tool that measures overheads associated with TM, designed to dissect these overheads and direct optimization efforts for the TM stack.

These tools uncover the subtle interaction of the TM system and the prefetching on BG/Q and help to study the implications for designing applications and choosing the TM mode of execution. Moreover, these tools enable to obtain a comprehensive understanding of the performance of synchronization mechanisms in *LULESH*, a Lagrange hydrodynamics proxy application, and find the cause for the missing performance with TM.

The third challenge is to correlate the gathered TM characteristics with microarchitecture events for the optimization of TM applications using STMs. Although this correlation is not new by itself, it is extremely helpful and has not been researched extensively in this context. Besides choosing well-suited parameters to monitor the microarchitecture, the correct interpretation of the obtained values is of key importance. To simplify the optimization for humans, the Visualization Of TM Applications (VisOTMA) framework is developed that visualizes these traces and enables the programmer to identify frequently conflicting transactions and the corresponding values of microarchitecture parameters in a post-processing step.

The visualization of this aggregated data needs to be achieved in a way that an inexperienced as well as an experienced programmer can identify pathological execution patterns – posing

the fourth challenge. So far optimizations have been carried out by TM experts with excellent knowledge of the underlying TM system. An inexperienced programmer does not have or may not be willing to acquire the knowledge of the TM system. Thus, the inexperienced programmer follows a trial-and-error strategy to optimize the TM application. To speedup this process, we invent EigenOpt – an exploration tool based on EigenBench – as part of the VisOTMA framework. With the help of EigenOpt, any programmer can capture the TM characteristic of the application in terms of parameters for Eigenbench. These parameters combined with Eigenbench are straightforwardly used to explore the space available through optimizations. With this tool, unrewarding optimization directions can be excluded without modifying the application. In this thesis, we will research how to identify and avoid optimization attempts with diminishing returns. This will speedup the optimization process for an inexperienced programmer and also yield new insights for an experienced one.

The fifth challenge is to detect and exploit a potential phase behavior of TM applications and integrate this analysis in the VisOTMA framework. In case the behavior of the TM application has periods with high and low conflict probability, this behavior of the application can be detected and exploited. Exploiting these phases is motivated through the different proposed TM designs: optimistic conflict detection schemes detect conflicts at commit-time whereas pessimistic schemes check for conflicts at encounter-time. In the optimistic case, a conflict early in the execution of a transaction is noticed at commit time so that computations are performed that have to be undone. The wasted work in this transaction can be reduced, when switching from the optimistic to the pessimistic scheme. In this thesis, we transfer algorithms, Signal Analysis and Wavelet-based classification, that have been proposed for phase detection in other contexts, to TM. These enable the offline detection of a TM phase behavior.

To complement the information retrieved in the above challenges, we gather and interpret static information. First, we design and implement initial support for Transactional Memory in GCC. This freely available support may provide the baseline for a wide-spread adoption of TM. The original GTM design, a design for the integration of TM in the programming language `C` with GCC, is presented, implemented, and evaluated. Second, we research how to exploit static information inside of a compiler to select suited STM parameters to project the run time behavior of that TM application and give advice to the application developer. This approach is called MAPT for analysis of Memory Access Patterns in Transactions and helps to select an STM parameter at compile-time.

In short the main contributions of this thesis are the following:

- a tracing methodology for STM that captures the genuine TM application's behavior – in a way that generating traces has a lower influence on the TM application and a higher throughput than a comparable Pin tool – while at the same time optionally employing compression algorithms,
- a low-overhead tracing solution for hybrid TM that exploits the properties of the TMbox architecture to achieve low-intrusiveness and enables a guided optimization process through visualization that yields a relative performance gain of $24.1\,\%$ when moving from STM-only to a hybrid-ETL variant on TMbox,
- a set of best practices that describes how to use TM on IBM's new Blue Gene/Q architecture with HTM support. Applying these practices yields a speedup of 1.22 over a simple transactional version of a Monte Carlo Benchmark and a speedup of

4.4 for an optimized TM version of a Smoothed Particle Hydrodynamics method from the PARSEC suite over a simple TM version. Additionally three tools are specifically designed for the evaluation and optimization of TM performance that highlight the interaction of TM and prefetching on BG/Q and help to identify the cause for the missing performance with TM in a Lagrange hydrodynamics proxy application,

- a novel framework for the optimization of STM applications (VisOTMA) that provides the following additional features
  - visualization of TM applications to identify pathological behavior that can help to tune the transaction's size in a way that the tuned TM version of a simulated fluid flow benchmark, implemented with a smoothed particle hydrodynamics method, yields a speedup of 1.43 over the initial TM version,
  - correlation of TM characteristics with microarchitecture events to better steer the optimization process and gain insights in the run time behavior and the applicability of STM to two variants of the method of Conjugate Gradients and reveal details whether the changed utilization of the microarchitecture is due to an altered convergence behavior or the choice of a different algorithmic formulation while at the same time comparing TM to other means of synchronization,
  - EigenOpt, an exploration tool that speeds up the optimization process for inexperienced programmers and excludes directions with diminishing returns which also helps experienced programmers,
  - algorithms for the detection of phase behavior in TM applications uncovering the additional potential of exploiting the resulting phase changes through adaptation of the TM system,
- design, implementation, and evaluation of inital support for TM in the GCC compiler,
- a new approach that gathers and exploits static information to select suited STM parameters and speedup the run time of the application yielding a relative improvement in execution time of 14.7 % for a transactional K-means clustering algorithm and 16.9 % for learning a Bayesian network implemented with transactions.

Zusammenfassung

Transactional Memory (TM) wurde als eine Architekturerweiterung vorgeschlagen, die die Verwendung von Datenstrukturen ohne Sperren ermöglichen soll. Durch die Allgegenwart von heutigen Mehrkernsystemen bekommt diese Idee neuen Schwung. Als Motivation für die Erfindung von TM diente die zu vereinfachende Synchronisation von mehreren parallel ablaufenden Fäden in einem System mit gemeinsam verwendetem Speicher. TM unterstützt hierbei einen optimistischen Ablauf der Transaktionen, welcher im Gegensatz zum pessimistischen Ablauf keinen wechselseitigen Ausschluss der Fäden erzwingt und somit eine parallele Abarbeitung unter der Bedingung, dass keine Wettlaufsituation auftritt, ermöglicht. Im Fall einer solchen Wettlaufsituation, die zum Beispiel dadurch entstehen kann, dass zwei Transaktionen auf die gleiche Speicherstelle schreiben, führt dies zu einem Konflikt der beiden Transaktionen, der erkannt und aufgelöst werden muss. Hierfür gibt es ein TM-Laufzeitsystem, welches die Zugriffe auf gemeinsam verwendeten Speicher innerhalb von Transaktionen überwacht. Diese Laufzeitsysteme für TM können in Software, kurz STM, Hardware, kurz HTM, oder als Hybridsystem realisiert werden. Die meisten der Forschungssysteme im Bereich von TM sind spezialisiert auf Spracherweiterungen, Unterstützung durch Übersetzer oder die Optimierung von algorithmischen Details des TM-Systems. Jedes der resultierenden TM-Systeme besitzt eine unterschiedliche Leistungscharakteristik und eine Vielzahl von Parametern, die abgestimmt werden können. Der hieraus resultierende Optimierungsraum überwältigt den Anwendungsprogrammierer, der jedoch das Zielpublikum für die Einführung von TM darstellte. Im Gegensatz zu anderen Forschungsaktivitäten beschäftigt sich diese Arbeit mit Werkzeugen für TM, welche die Lücke zwischen dem Anwendungsprogrammierer und dem Entwickler des TM-Systems überbrücken sollen. Diese Werkzeuge ermöglichen Einblicke in das Laufzeitverhalten einer Anwendung mit TM und leiten den Anwendungsentwickler zur Optimierung der TM-Anwendung an.

Im Gegensatz zu anderen verwandten Arbeiten, die die Werkzeuge auf eine bestimmte Kombination von Programmiersprache und TM-System zuschneiden, präsentiert diese Arbeit Lösungen für mehrere TM-Systeme (Software, Hardware und hybride) und bezieht das Setzen von unterschiedlichen Parametern für jedes dieser TM-Systeme mit ein. Zudem verwenden die vorgeschlagenen Methoden und Verfahren zur Optimierung von TM-Anwendungen Informationen von allen Schichten des Software-Systems. Statische Informationen, Informationen über das Laufzeitverhalten der Anwendung, sowie Expertenwissen müssen extrahiert und gesammelt werden, um diese neuen Methoden und Strategien zur Optimierung von TM-Anwendungen zu entwickeln. Das Erheben und Verwenden der benötigten Informationen verlangt nach der Bewältigung der folgenden Herausforderungen.

Als erste Herausforderung muss das unverfälschte Laufzeitverhalten der TM-Anwendung festgehalten werden. Dieses ist von hoher Bedeutung, weil Transaktionen sehr empfindlich auf eingeführte Verzögerungen eines Fadens reagieren, wodurch künstliche Konflikte

mit anderen Transaktionen ausgelöst werden können. Hierdurch kann das aufgezeichnete Anwendungsverhalten durch die Aufzeichnungsmechanismen selbst verfälscht werden. Folglich muss eine leichtgewichtige Lösung für das Erstellen von Spurdateien, verbunden mit einer geringen Verfälschung der Anwendung, entwickelt und ausgewertet werden. Diese Arbeit präsentiert ähnliche Lösungen für STM- und hybride TM-Systeme, welche Spurdateien der Ereignisse der TM-Anwendung erzeugen. Für das STM-System wird für die Aufzeichnung der häufig vorkommenden Ereignisse, wie beispielsweise transaktionale Lese- und Schreibzugriffe, die Bandbreitenbeschränkung der Festplatte schnell zum Flaschenhals. Um diesem entgegenzuwirken, muss eine Reduktion der Daten, die auf die Festplatte geschrieben werden, erreicht werden. Hierfür erforschen wir wie diese Ereignisse zur Laufzeit komprimiert werden können, ohne die Anwendung damit zu stören. Ein mehrfädiges Kompressionsschema für Spuren von TM-Ereignissen wird entworfen, implementiert und ausgewertet. Wir erweitern TinySTM, eine Wort-basierte, quelloffene STM-Implementierung mit der Infrastruktur zum Erstellen von TM-Ereignisspuren. In Folge des leichtgewichtigen und komprimierenden Schemas für Spuren von TM-Ereignissen, generiert die resultierende TracingTinySTM Ereignis-Dateien, die das unverfälschte Verhalten der TM-Anwendung wiedergeben. Eine Lösung zur Spurerstellung mit geringem Mehraufwand wird für ein FPGA-basiertes hybrides TM-System, genannt TMbox, demonstriert. Jedes Rechenelement wird um eine Überwachungseinheit erweitert, die bei der Ausführung interessanter Instruktionen ein entsprechendes Ereignis generiert. Falls eine überwachte Instruktion auftritt, wird ein Zeitstempel generiert und gemeinsam mit dem Ereignis an die Aufzeichnungseinheit weitergegeben. Diese Einheit benutzt Leerzeiten auf dem Bus, um diese Ereignisse an den Speicher des Hauptcomputers zu schicken. Obwohl diese Methoden auf den speziellen Kontext von TM zugeschnitten sind und dort angewendet werden, sind sie universell und könnten auch in einem anderen Kontext angewendet werden. Andere Anwendungsgebiete sind das Überwachen von beliebigen Ereignissen in einem FPGA-basierten Prozessor-Prototypen oder das Aufzeichnen und Komprimieren von Ereignissen in einer Mathematikbibliothek.

Die zweite Herausforderung entsteht mit der Veröffentlichung der Blue Gene/Q Architektur, welche eine Unterstützung für HTM besitzt, durch IBM. Wir erstellen eine Menge an bewährten Vorgehensweisen, die es dem Anwendungsentwickler ermöglichen, das volle Potential der Architektur inklusive des TM-Systems zu nutzen. Zu diesem Zweck führen wir einen neuen Benchmark, genannt CLOMP-TM, ein, welcher über eine Vielzahl an Parametern verfügt, die es ermöglichen Transaktionen mit unterschiedlichen Granularitäten, Konfliktraten und verschiedenen Rechenkernen, welche im wissenschaftlichen Rechnen Verwendung finden, zu erforschen. Dieses ermöglicht neue Einsichten durch die Evaluation des TM-Systems und den Vergleich der Leistung mit anderen Synchronisationprimitiven, welche von OpenMP unterstützt werden. Zudem benötigt diese neue BG/Q Architektur auch TM-spezifische Werkzeugunterstützung, um die Optimierung von wissenschaftlichen Anwendungen zu ermöglichen. Eine Lösung, die auf der Erzeugung von Spuren von TM-Ereignissen basiert, um das Laufzeitverhalten festzuhalten, ist für dieses proprietäre System nicht bzw. nur eingeschränkt durchführbar. Das HTM-System benötigt eine vollständige Softwareunterstützung, die Übersetzer und Laufzeitsystem umfasst. Folglich entwerfen, implementieren und bewerten wir drei verschiedene Werkzeuge für TM, welche es Programmierern ermöglichen, die Feinheiten der Ausführung mittels TM zu ergründen und tragen damit Folgendes zum Stand der Forschung bei:

- das erste Werkzeug, welches das Erstellen von Laufzeitprofilen von Anwendungen mit MPI, OpenMP und TM auf der BG/Q Architektur ermöglicht,

- ein Werkzeug für TM, welches durch Ereignisspuren von TM Statistiken in Kombination mit der Auslastung der Architektur, welche auf der Granularität des ausführenden Fadens erhoben werden, eine detaillierte Analyse mittels des Visualisierungswerkzeugs Vampir, das dem Stand der Technik entspricht, ermöglicht und
- ein Werkzeug, das den Mehraufwand, der mit TM einhergeht, misst, den jeweiligen Ausführungsphasen einer Transaktion zuordnet und damit ermöglicht, dass Optimierungsversuche direkt auf die relevanten Teile der Softwareschichten abzielen können.

Diese Werkzeuge enthüllen die Wechselwirkung zwischen TM-System und dem Vorabladen von Werten aus dem Speicher auf der BG/Q Architektur und helfen dabei, die Auswirkungen von Entwurfsentscheidungen für die Anwendung und die Auswahl des TM Ausführungsmodus zu verstehen. Darüber hinausgehend ermöglichen diese Werkzeuge ein umfassenderes Verständnis der Synchronisationsmechanismen in *LULESH*, einer Lagrange-Hydrodynamik-Stellvertreter-Anwendung und identifizieren den Grund für die fehlende Leistung mit TM.

Die dritte Herausforderung ist die Korrelation der gesammelten TM-Charakteristiken mit Ereignissen, die das Verhalten der Mikroarchitektur beschreiben, um dieses zur Optimierung von STM-Anwendungen zu verwenden. Obwohl diese Art der Korrelation selbst nicht neu ist, ist sie extrem hilfreich und wurde in diesem Kontext noch nicht weitergehend untersucht. Die korrekte Interpretation der erhobenen Werte ist neben der Auswahl von geeigneten Parametern der Mikroarchitektur, welche dann überwacht werden, von höchster Bedeutung. Um die Optimierung für Menschen zu vereinfachen, entwickeln wir das Rahmenwerk VisOTMA zur Visualisierung von TM-Anwendungen. Dieses ermöglicht es in einem Nachbearbeitungsschritt dem Programmierer häufig konfligierende Transaktionen und die zugehörigen Werte der Mikroarchitekturparameter zu identifizieren.

Die Visualisierung dieser aggregierten Daten muss so erfolgen, dass sogar ein unerfahrener Programmierer pathologische Ausführungsmuster erkennen kann. Dies stellt die vierte Herausforderung dar. Bisher wurden Optimierungen von TM-Experten vorgenommen, die ein ausgezeichnetes Wissen über das zugrundeliegende TM-System hatten. Ein unerfahrener Programmierer hat dieses Wissen über das TM-System nicht oder will es sich nicht aneignen. Folglich verfolgt dieser bei der Optimierung der TM-Anwendung eine Strategie, die auf Versuch und Irrtum beruht. Um diesen Prozess zu beschleunigen, führen wir EigenOpt ein. EigenOpt ist ein Erkundungswerkzeug, welches auf EigenBench beruht und in das VisOTMA-Rahmenwerk integriert wird. Mit der Hilfe von EigenOpt kann jeder Programmierer das charakteristische TM-Laufzeitverhalten in Form von Parametern für EigenBench festhalten. Diese Parameter in Verbindung mit EigenBench werden kanonisch verwendet, um den Raum für Optimierungen zu erforschen. Mit diesem Werkzeug können Optimierungsansätze, die keinen Erfolg versprechen, ausgeschlossen werden, ohne die Anwendung zu modifizieren. In dieser Arbeit werden wir erforschen, wie man das charakteristische TM-Anwendungsverhalten automatisiert festhält und Optimierungsansätze ohne Aussicht auf Gewinn identifiziert und vermeidet. Dies wird den Optimierungsvorgang für einen unerfahrenen Programmierer beschleunigen und neue Einsichten für einen erfahrenen Programmierer bereithalten.

Die fünfte Herausforderung besteht in der Detektion und dem Ausnutzen eines potentiellen Phasenverhaltens der TM-Anwendung und in der Integration dieser Analyse in das bestehende Rahmenwerk VisOTMA. Für den Fall, dass das TM-Anwendungsverhalten Perioden

mit hohem und niedrigem Konfliktpotential aufweist, kann dieses Anwendungsverhalten erkannt und eventuell ausgenutzt werden. Die Ausnutzung dieses Phasenverhaltens wird motiviert durch die verschiedenen TM-Entwürfe: eine optimistische Konflikterkennung erkennt Konflikte erst zur Zeit des Abschlusses der Transaktion, während eine pessimistische Konflikterkennung beim Zugriff auf die Daten bereits Konflikte erkennt. Wenn im optimistischen Fall ein Konflikt früh in der Ausführung einer Transaktion auftritt und erst beim Abschluss der Transaktion erkannt wird, werden weitere Berechnungen angestellt, die zum Ende der Transaktion wieder rückgängig gemacht werden müssen. Diese verschwendete Arbeit in dieser Transaktion kann verringert werden, wenn man von dem optimistischen zum pessimistischen Schema wechselt. In dieser Arbeit übertragen wir Algorithmen wie die Signal Analyse und eine Klassifikation basierend auf Wavelets, welche für die Phasenerkennung in anderen Bereichen vorgeschlagen wurden, auf TM. Diese Algorithmen ermöglichen die Erkennung eines Phasenverhaltens in TM.

Um die Informationen, die in den vorhergehenden Herausforderungen erhoben wurden zu ergänzen, sammeln und interpretieren wir auch statische Informationen. Zuerst entwerfen und implementieren wir eine Unterstützung von Transactional Memory im GCC. Diese frei verfügbare Unterstützung, genannt GTM, kann als Basis für einen weitverbreiteten Einsatz von TM dienen. Der ursprüngliche Entwurf von GTM wird präsentiert, implementiert und ausgewertet. Zweitens erforschen wir, wie man statische Informationen innerhalb des Übersetzers verwenden kann, um geeignete Parameter einer STM-Bibliothek anhand des erwarteten Laufzeitverhaltens der TM-Anwendung auszuwählen und den Anwendungsprogrammierer damit zu unterstützen. Dieser Ansatz wird von uns MAPT genannt, da er die Speicherzugriffsmuster in Transaktionen analysiert und bei der Selektion eines STM-Parameters zur Übersetzungszeit unterstützt.

Zusammengefasst sind die Hauptbeiträge dieser Arbeit die folgenden:

- eine Methodik zur Spurerstellung von STM-Ereignissen, die das unverfälschte TM-Anwendungsverhalten in einer Form festhält, so dass diese einen geringeren Einfluss auf die TM-Anwendung und einen höheren Durchsatz zur Folge hat als ein vergleichbares Werkzeug, welches auf dem dynamischen Binärinstrumentierer Pin basiert, wobei bei unserem Ansatz optional Kompressionsalgorithmen zum Einsatz kommen,
- eine Lösung zur Spurerstellung von Ereignissen in hybriden TM-Systemen, die wenig Overhead erzeugt und Eigenschaften der TMbox Architektur ausnutzt, um eine geringe Beeinflussung und eine Anleitung zum Optimierungsprozess mittels Visualisierung bereitstellt, die einen relativen Leistungsgewinn von $24.1\,\%$ zur Folge hat, wenn man von einer Ausführung in STM zu einer hybrid-ETL Variante auf TMbox wechselt,
- eine Menge an bewährten Vorgehensweisen, die beschreiben, wie man das neue HTM-System der Blue Gene/Q Architektur verwendet. Die Anwendung dieser Vorgehensweisen zeigt einen Gewinn von 1.22 im Vergleich zu einer einfachen transaktionalen Version eines Monte-Carlo-Benchmarks und einen Gewinn von 4.4 für eine optimierte TM-Version einer geglätteten Partikelmethode der Hydrodynamik aus der PARSEC-Sammlung. Zusätzlich werden drei Werkzeuge speziell für die Auswertung und Optimierung der Leistung von TM entworfen, so dass diese die Interaktionen zwischen TM-System und dem Vorabladen von Werten aus dem Speicher auf der BG/Q beleuchten und helfen den Grund für die fehlende Leistung von TM in einer Lagrange-Hydrodynamik-Stellvertreter-Anwendung zu identifizieren,

- ein neues Rahmenwerk zur Optimierung von STM-Anwendungen (VisOTMA), welches die folgenden zusätzlichen Eigenschaften besitzt:
  - die Visualisierung des TM-Anwendungsverhaltens ermöglicht, um pathologisches Verhalten der TM-Anwendung identifizieren zu können und zusätzlich hilft die Transaktionsgröße in einer Art und Weise anzupassen, dass die optimierte STM-Variante einer geglätteten Partikelmethode der Hydrodynamik aus der PARSEC-Sammlung einen Laufzeitgewinn von 1.43 gegenüber der initialen TM-Version erzielt,
  - die Korrelation von TM-Charakteristika und Ereignissen der Mikroarchitektur, welche es ermöglicht, den Optimierungsprozess besser zu steuern und hierfür Einblicke in das Laufzeitverhalten und die Anwendbarkeit von STM gewährt. Diese Methode wird auf zwei Varianten des mathematischen Verfahrens der konjugierten Gradienten angewendet und gibt detailliert Aufschluss darüber, welche Veränderungen der Nutzung der Mikroarchitektur durch ein verändertes Konvergenzverhalten und welche durch andere Effekte der Umformulierung des Verfahrens bedingt sind, wobei zusätzlich TM mit anderen Möglichkeiten zur Synchronisation verglichen wird,
  - EigenOpt, ein Erkundungswerkzeug, welches die Optimierung sogar für unerfahrene Entwickler beschleunigt und Optimierungsansätze, die keinen Erfolg versprechen, ausschließt, ohne dass die Anwendung modifiziert werden muss,
  - Algorithmen für die Erkennung eines Phasenverhaltens in TM-Anwendungen, die zusätzliches Potential zur Ausnutzung der Phasenwechsel durch die Anpassung des TM-Systems aufzeigen,
- den Entwurf, die Implementierung und die Auswertung der initialen Unterstützung des Übersetzers für TM im GCC,
- einen neuen Ansatz, der statische Informationen sammelt und ausnutzt, um einen passenden Parameter der STM-Bibliothek auszuwählen und einen Laufzeitgewinn der Anwendung mit einer relativen Steigerung von $14.7\,\%$ für einen transaktionalen K-means Bündelungsalgorithmus und $16.9\,\%$ für das Lernen eines Bayes'schen Netzwerks mittels Transaktionen erzielt.

Contents

List of Publications Relevant for this Thesis

[HJK+13] Vincent Heuveline, Sven Janko, Wolfgang Karl, Björn Rocker, and Martin Schindewolf. Software Transactional Memory, OpenMP and Pthread Implementations of the Conjugate Gradients Method – A Preliminary Evaluation. In Michel Daydé, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 300–313. Springer Berlin / Heidelberg, July 2013.

[KSS+12] Philipp Kirchhofer, Martin Schindewolf, Nehir Sonmez, Oriol Arcas, Osman S. Unsal, Adrian Cristal, and Wolfgang Karl. Enhancing an HTM System with Monitoring, Visualization and Analysis Capabilities. In *Euro-TM Workshop on Transactional Memory (WTM 2012)*, April 2012. Abstract available at `http://www.eurotm.org/action-meetings/wtm2012/program/abstracts#Kirchhofer`.

[SAK+12] Nehir Sonmez, Oriol Arcas, Philipp Kirchhofer, Martin Schindewolf, Osman S. Unsal, Adrián Cristal, and Wolfgang Karl. A low-overhead Profiling and Visualization Framework for Hybrid Transactional Memory. In *FCCM 2012: The 20th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8, 2012. `http://fccm12.cse.sc.edu/4699a001.pdf`.

[SBG+12] Martin Schindewolf, Barna Bihari, John Gyllenhaal, Martin Schulz, Amy Wang, and Wolfgang Karl. What Scientific Applications Can Benefit from Hardware Transactional Memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 90:1–90:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[SCK+09] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, and Luca Benini. Towards Transactional Memory Support for GCC. In *First International Workshop on GCC Research Opportunities*, GROW '09, January 2009. Held in conjunction with: the fourth International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC).

[SEK11] Martin Schindewolf, Alexander Esselson, and Wolfgang Karl. Compiler-Assisted Selection of a Software Transactional Memory System. In Mladen Berekovic, William Fornaciari, Uwe Brinkschulte, and Cristina Silvano, editors, *Architecture of Computing Systems - ARCS 2011*, volume 6566 of *Lecture*

Notes in Computer Science, pages 147–157. Springer Berlin / Heidelberg, 2011.

[SK09] Martin Schindewolf and Wolfgang Karl. Investigating Compiler Support for Software Transactional Memory. In *Proceedings of ACACES 2009 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*, pages 89–92, Terrassa, Spain, July 2009. Academia Press, Ghent.

[SK12] Martin Schindewolf and Wolfgang Karl. Capturing Transactional Memory Application's Behavior – The Prerequisite for Performance Analysis. In *International Conference on Multicore Software Engineering, Performance and Tools (MSEPT 2012)*, volume 7303 of *Lecture Notes in Computer Science*, pages 30–41. Springer Verlag, May 31–June 1, 2012.

[SRKH13] Martin Schindewolf, Björn Rocker, Wolfgang Karl, and Vincent Heuveline. Evaluation of two Formulations of the Conjugate Gradients Method with Transactional Memory. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *19th International European Conference on Parallel and Distributed Computing Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2013. Accepted for publication.

[SSB+12] Martin Schindewolf, Martin Schulz, Barna Bihari, John Gyllenhaal, Amy Wang, and Wolfgang Karl. Performance Analysis of and Tool Support for Transactional Memory on BG/Q. In *Euro-TM Workshop on Transactional Memory (WTM 2012)*, April 2012. Abstract available at `http://www.eurotm.org/action-meetings/wtm2012/program/abstracts#Schindewolf`.

List of Figures

List of Tables

Listings

1. Motivation for Tools targeting Transactional Memory

1.1 Transactional Memory for Parallel Programming

With multi-core processors becoming the norm for desktop as well as server machines, parallel programming gains importance in everyday software development. Due to the difficulties associated with conventional locking, Herlihy and Moss proposed Hardware Transactional Memory (HTM) to facilitate the synchronization in applications using shared memory [91]. The concept of transactions is borrowed from database systems and releases the programmer from the burden of managing low-level primitives e.g., locks to synchronize concurrent threads of execution.

For programmers, Transactional Memory offers the convenient abstraction of an atomic block, also called transaction, to synchronize concurrent accesses to shared memory. A transaction may contain a finite sequence of instructions that execute with the following properties: Atomicity, Consistency, and Isolation (ACI) [83]. These ACI properties provide a convenient abstraction for coordinating concurrent accesses to shared data and herewith simplify parallel programming. TM delivers the means to overcome most of the shortcomings of traditional lock-based synchronization, such as priority inversion, deadlock, livelock, and convoying.

Moreover, TM features optimistic concurrency that replaces the mutual exclusion of traditional synchronization. This enables to execute transactions in parallel and detect conflicting memory accesses of different threads. A run time system, implemented in hardware (HTM), software (STM) or both (hybrid TM), implements the conflict detection and maintains speculative versions of the data.

The optimistic concurrency with TM nourishes the hope for an increased scalability compared with pessimistic synchronization. Scalability improvements gain importance with the ever increasing core counts of the new multi-core generations. Another advantage of TM is the composability of independently developed parallel libraries. This may facilitate the integration of third party libraries in the development cycle of parallel software.

Moreover, the transaction simplifies to reason about the correctness of the parallel code because transactions are simple to understand and insert. Thus, TM improves parallel

programming over lock-based synchronization, which especially attracts programmers new to the development of parallel software.

1.2 The Missing Performance with TM

The target audience judges whether TM is a success or not. This audience may have more demands on TM than the expected gains in usability. For developers who start to program in parallel, a convenient abstraction that simplifies synchronization may be convincing. For experts in high performance computing (HPC), this may not suffice. In this use case, the composability of parallel modules and the expected gains when maintaining the software are advantageous but the performance of TM is also a very important factor.

The performance of TM depends on the implementation of the run time system. STM comes with known performance deficiencies that originate from the overhead due to the instrumentation of the memory accesses, executing the algorithm for conflict detection, lock acquisition and release, and validation of the read and write sets. HTMs often have limitations e.g., in the number of memory locations that can be accessed inside of a transaction or the type of instructions that can be executed. Otherwise the transaction fails and retries. Hybrid TM inherits the performance of both systems so that a transition from hardware to software execution yields a performance degradation.

Although these overheads and limitations are known, the application developer does not necessarily know whether overheads cause the unexpectedly low performance or an execution pattern that results in repeated rollbacks of a transaction. The issue is that the application developer is unaware of the TM application's behavior and has no profound basis to take counter measures e.g., to reorganize the transactions, choose a better suited TM run time system or reorganize the data structures to achieve a better utilization of the microarchitecture.

Moreover, there has been little research on the human factor in TM programming so that the granularity of transactions and the introduced costs through inappropriate use of transactions have not been quantified. Tools for TM can detect and help ameliorate these cases so that even inexperienced programmers can detect pathological execution patterns in their TM application and judge whether this behavior negatively influences the performance of the TM application.

1.3 Tool Support for Simplifying the Optimization of TM Programs

So far tool support for TM has been proposed to support STM [7, 191, 127, 123, 220, 29, 76] or HTM [31, 71]. While these solutions are tailored to a specific TM system and programming language, we present the first holistic approach that considers multiple TM systems (Software, Hardware, and hybrid TM) as well as includes information from more than two layers of the TM software stack. Information from the TM application, TM compiler, TM run time system, and hardware pose the inputs for the novel methods and strategies for the optimization of TM applications presented in this thesis. We collect information from these different layers of the TM software stack in order to obtain a broader view on the performance of the application and advance the state-of-the-art. The additional information is expected to yield higher performance gains and enable optimizations beyond

that of previous tools for TM that consider only one or two layers of the TM software stack. In particular we identify the following possibilities to improve the state-of-the-art in tools for TM.

The following related works use profiling or sampling information from the TM run time system that is retrieved during run time and combine it with information from the source code [7, 191, 127, 220, 123, 29, 76]. The combination of information from the TM run time and source code is important and valuable for the programmer, but cannot explain performance deficiencies in the presence of a TM behavior that shows very few rollbacks. In order to sufficiently explain the performance of the TM system in these cases and provide complementary information from the microarchitecture for all cases, we research how to use information from the hardware performance counters of the microarchitecture. These counters enable to rate and compare the utilization of resources of the microarchitecture e.g., L1 caches and help to identify the sources for performance degradation with TM even for a TM application with very few rollbacks. Moreover, readings from these performance counters also help to compare synchronization using TM with other synchronization constructs e.g., with OpenMP primitives and enable new insights into the interaction of the TM system with the microarchitecture that have not been possible with previous tools for TM.

Optimization of TM applications with the mentioned tools is a trial-and-error process that takes the following steps: first, profile the TM application, then find the data structures that are heavily contended and identify the corresponding transactions, use the profiling information to determine the *wasted work* caused by transactions that abort and either rearrange or substitute the data structures or change the reordering of statements in the transaction to reduce the amount of wasted work [220]. Hence, an application developer must invest time and effort to refactor the data structures and/or transactions of the TM application in order to reduce the amount of wasted work. A new version of the application must be tested for correctness, executed and compared with the previous profile in order to find out whether the code changes yield the expected performance gains. Often this a disappointing and tiring process that yields diminishing returns. We seek to alleviate this process through providing a set of best practices for the programming with TM. These practices enable the programmer to design the TM application according to criteria that incorporate transactional length and conflict potential. With these practices an application developer easily sees whether a transaction meets the requirements or not and can direct the optimization efforts so that the transactions in the TM application meet the requirements formulated in the best practices. A second approach researches how to use hardware performance counters to extract the characteristic parameters of the TM application behavior. With these characteristic parameters, a tool such as the EigenBench microbenchmark [94] helps to simulate the effects of possible optimizations on the target application and helps to exclude optimization directions with diminishing returns. This approach may help application developers to simulate a set of optimizations and select and transfer the most promising one to the actual TM application.

Moreover, the existing tools do not account for the time varying behavior of TM applications. In order to make the time varying behavior of a TM application easily accessible for an inexperienced as well as an experienced application developer, transitions between execution phases with low and high contention between transactions must be recognized automatically with a tool. This approach helps to assess the potential of optimizations that exploit the transition between optimistic and pessimistic conflict detection modes of an

STM system. In order to identify a possible optimization possibility and to answer the question whether phases exist in current TM applications, we develop a post-processing tool that analyzes the execution phases of a TM application and detects phase changes. This tool enables even inexperienced application developers to identify execution phases in the application and determine the number of phase changes to judge the optimization potential.

Until now, tools for TM do not take advantage of the static information from the TM application that is available throughout the compilation process. This may lead to missed optimization opportunities and result in selecting a standard parameter setting of the STM system that may not be the best for this particular application. Hence, incorporating static information in the flow of the optimization of TM applications seems an idea worth to research in this thesis.

Therefore, static information, information about the run time behavior and best practices, that condense the knowledge of experts, need to be extracted to develop these new methods and strategies for the optimization of TM applications. The practical result of researching these methods and strategies are tools for TM that can be easily applied even by inexperienced application developers. These tools extract the run time behavior of a TM application, provide means, e.g., a representative application with a multitude of input parameters, to find best practices, support the visualization of the run time behavior, and thus help experienced as well as inexperienced application developers to optimize the TM application. Extracting and combining information from TM systems implemented in hardware, software or both in a suitable way poses the following six challenges that we address in this thesis.

The first challenge is to capture the genuine TM application's behavior at run time. This is especially important because transactions are sensitive to artificially introduced delays of a thread which may lead to an introduced TM conflict. This may lead to a recorded application behavior that is biased through the tracing machinery. Therefore, a lightweight trace generation scheme with a small probe-effect needs to be developed and evaluated. This thesis presents similar solutions for STM and hybrid TM that generate event traces. For STM, the logging of frequently occurring events, such as TM loads and stores, quickly saturates the write bandwidth of the hard disk. Thus, a reduction of the amount of data to be written to hard disk needs to be achieved. Therefore, we research how these events can be compressed online without disturbing the application. A multi-threaded trace compression scheme is designed, implemented, and evaluated. We enhance the TinySTM, a word-based open source STM, with tracing facilities. Due to the lightweight and compressing tracing scheme the resulting TracingTinySTM generates event traces that capture the genuine TM application's behavior. For an FPGA-based hybrid TM system, called TMbox, a low-overhead tracing solution is shown. Each processing element is enhanced with an event generation unit for monitoring. When an instruction of interest is encountered, a corresponding time-stamped event is generated and passed on to the log unit. This unit uses idle times on the bus to transfer it to the memory of the host machine. The event tracing methodology captures transactions that utilize the HTM part of the hybrid TM system as well as transactions using the STM part. Our approach is the first to capture the behavior of a TM application in a hybrid TM system. Although all methods are tailored to and applied in the context of TM, they are universally applicable in a different context. Example application areas are the monitoring of arbitrary events in an FPGA-based processor prototype or the logging and compressing of events in a math library.

The second challenge arises with the release of IBM's new Blue Gene/Q architecture with HTM support. We establish a set of best practices for the new architecture so that the application developer can exploit the full potential of the architecture including the TM subsystem. For this purpose, we introduce a new benchmark, CLOMP-TM, that has a series of parameters that allow us to explore varying transaction granularities and conflict rates, coupled with typical computational kernels found in scientific codes. This enables new insights through evaluating the TM system and comparing the performance with other synchronization primitives of OpenMP. Further, this new BG/Q architecture also requires tool support to enable optimizations of scientific applications. A trace-based solution to capture the run time behavior on a per event basis is not feasible for this proprietary HTM system. This HTM system requires a proprietary software stack, including compiler and run time system. Thus, we design, implement and evaluate three different tools for TM that enable programmers to explore the subtleties of TM execution. We contribute the first tool that profiles applications using MPI and OpenMP with TM on BG/Q, a tracing tool for TM that enables in-depth inspection of thread-level execution and utilization of the architecture through visualization with the state-of-the-art visualization tool Vampir, and a tool that measures overheads associated with TM, designed to dissect these overheads and direct optimization efforts for the TM stack. These tools incorporate the use of BG/Q specific hardware performance counters in order to uncover the subtle interaction of the TM system and the prefetching on BG/Q and help to study the implications for designing applications and choosing the TM mode of execution. Moreover, these tools enable to obtain a comprehensive understanding of the performance of synchronization mechanisms in *LULESH*, a Lagrange hydrodynamics proxy application, and find the cause for the missing performance with TM.

The third challenge is to correlate the gathered TM characteristics with microarchitecture events for the optimization of TM applications using STMs. Although this correlation is not new by itself, it is extremely helpful and has not been researched extensively in this context. Besides choosing well-suited parameters to monitor the microarchitecture, the correct interpretation of the obtained values is of key importance. To simplify the optimization for humans, the Visualization Of TM Applications (VisOTMA) framework is developed that visualizes these traces and enables the programmer to identify frequently conflicting transactions and the corresponding microarchitecture parameters in a post-processing step.

The visualization of this aggregated data needs to be achieved in a way that an inexperienced as well as experienced programmer can identify pathological execution patterns – posing the fourth challenge. So far optimizations have been carried out by TM experts with excellent knowledge of the underlying TM system. An inexperienced programmer does not have or may not be willing to acquire the knowledge of the TM system. Thus, the inexperienced programmer follows a trial-and-error strategy to optimize the TM application. To speedup this process, we invent EigenOpt – an exploration tool based on EigenBench – as part of the VisOTMA framework. With the help of EigenOpt, any programmer can capture the TM characteristic of the application in terms of parameters for Eigenbench. These parameters combined with Eigenbench are straightforwardly used to explore the space available through optimizations. With this tool, unrewarding optimization directions can be excluded without modifying the application. In this thesis, we will research how to identify and avoid optimization attempts with diminishing returns. This will speedup the optimization process for an inexperienced programmer and yield new insights for an experienced one.

The fifth challenge is to detect and exploit a potential phase behavior of TM applications and integrate this analysis in the VisOTMA framework. In case the behavior of the TM application has periods with high and low conflict probability, this behavior of the application can be detected and exploited. Exploiting these phases is motivated through the different proposed TM designs: optimistic conflict detection schemes detect conflicts at commit-time whereas pessimistic schemes check for conflicts at encounter-time. In the optimistic case a conflict early in the execution of a transaction is noticed at commit time so that the programs performs computations that have to be undone. The wasted work in this transaction can be reduced, when switching from the optimistic to the pessimistic scheme. In this thesis, we transfer algorithms, Signal Analysis and Wavelet-based classification, that have been proposed for phase detection in other contexts, to TM. These enable the offline detection of a TM phase behavior.

To complement the information retrieved in the above challenges, we gather and interpret static information. First, we design and implement initial support for Transactional Memory in GCC. This freely available support may provide the baseline for a wide-spread adoption of TM. The original GTM design, a design for the integration of TM in the `C` programming language with GCC, is presented, implemented, and evaluated. Second, we research how to exploit static information inside of a compiler to select suited STM parameters to project the run time behavior of that TM application and give advice to the application developer. This approach is called MAPT for analysis of Memory Access Patterns in Transactions and helps to select an STM parameter at compile-time.

1.4 Organization of this Thesis

The structure of this thesis is the following. Chapter 2 introduces the basic concept of transactional memory and elaborates on the advantages with TM. We present the three basic types of realizing TM in software, hardware, and as hybrid variant and illustrate these with prototypes. Further, we discuss the implications of the memory model on programming with TM.

We present closely related works in Chapter 3 and discuss these works with our approach.

The overall concept of researching methods and strategies for the optimization of TM applications is shown in Chapter 4. After illustrating the TM stack and the information associated with each layer, we discuss the testbeds for realizing the methods and strategies as tools and evaluating the approach.

Chapter 5 shows two solutions for fine grained tracing of TM events in STM and hybrid TM. While the STM solution may also compress the event traces, the tracing scheme for hybrid TM adds hardware structures to the TM system that enable tracing with low overhead in execution time. We compare both tracing schemes to assess the value of adding hardware components to generate traces of TM events.

We illustrate our framework and its components for post-processing of the TM run time information in Chapter 6. The visualization component illustrates the TM run time behavior of the TM application and uncovers pathological execution patterns with TM. A simple reference application helps to find a suited transaction length for execution with STM. We apply these findings to optimize a benchmark from the PARSEC suite and a numerical application. Another component of the framework implements two algorithms to detect execution phases with different conflict potential in TM applications. We apply these to the

STAMP benchmarks and discuss the results. Then, we present the EigenOpt approach that helps the programmer to identify and exclude optimization directions with diminishing returns.

Chapter 7 presents the initial design and implementation of TM support in GCC, adds ideas for future work based on the compiler extension and documents its progress. Then we present a static approach for analyzing memory access patterns in transactions, use this information to select a property of the STM system, and present results from applying it to test cases and STAMP benchmarks.

In Chapter 8 we present the first experiences with HTM on the BG/Q architecture from an application's perspective. We introduce CLOMP-TM, a benchmark that is aimed at evaluating TM systems for scientific workloads. We condense the findings from the experiments with CLOMP-TM into a set of best practices and apply them to a realistic Monte Carlo Benchmark code and a Smoothed Particle Hydrodynamics method.

In Chapter 9 we complement the best practices with a set of tools that enable each programmer to explore the subtleties of TM execution on BG/Q. We present detailed multi-threaded overhead measurements of the TM subsystem dividing transactional execution into three separate phases, uncover the subtle interaction of the TM system and the prefetching unit, and study the implications for the design of the application. Further, we evaluate how to choose the TM mode, obtain a comprehensive understanding of the performance of synchronization mechanisms in *LULESH*, a Lagrange hydrodynamics proxy application, and find the cause for the missing performance with TM.

Chapter 10 concludes this thesis, summarizes the findings from designing and implementing tools for each TM system, reviews the results from applying the tools to TM applications and gives some ideas how to extend this work in the future.

2. Fundamentals

This chapter introduces the basic concept of Transactional Memory in Section 2.1, the advantages of synchronizing shared memory applications with TM, and discusses the implementation of the TM run time system. Section 2.2.1 discusses the implementation of TM with software primitives, Section 2.2.2 presents hardware support for TM, and Section 2.2.3 combines both to a hybrid TM system. In Section 2.3 we introduce the different memory models for TM, discuss the differences, describe the progress that converges to a single memory model for STM, and sketch the implications of a memory model on compiler instrumentation and possible optimizations. With respect to the work presented in this thesis, this chapter presents the basic concept of TM and discusses three implementation techniques in order to motivate the need for tools tailored to each of these implementation strategies and highlight the respective specialties of the implementation (e.g., memory model) that tools need to consider.

2.1 Transactional Memory Concept and Properties

Traditionally programmers use synchronization constructs that enforce mutual exclusion to ensure a correct parallel execution of multiple threads in a program with shared memory. Locks or critical sections are placed in the code and prevent that more than one thread enters this section at the same time. With Transactional Memory (TM), managing these locks becomes oblivious. Instead of low-level locking primitives, Transactional Memory (TM) offers the high-level concept of transactions. These transactions may consist of a structured block that contains control flow and function calls. The most important difference to traditional lock-based synchronization is – besides the ease-of-use – the optimistic concurrency with TM. Instead of enforcing mutual exclusion (as locks do), transactions execute in parallel and conflicting memory accesses are resolved by a TM run time system. By replacing potentially many locks with transactions, the application developer gains a convenient and straightforward way of handling synchronization in parallel shared memory programs. TM reliefs the programmer of managing locks explicitly and thus avoids many of the pitfalls associated with locking: deadlocks, priority inversion, convoying, and lack of scalability.

Two locking strategies with different granularities are commonly distinguished: fine-grained and coarse-grained locking. While fine-grained locking offers performance and

scalability, it is difficult to maintain and add new components to the software because lock acquire and release must follow a specific order. The application developer establishes and implements this virtual order because otherwise the program will deadlock. Other application developers working on the same application, now have to adhere to this order of lock acquires and releases although it may not be written down explicitly. Thus, programming errors in parallel programming on large software projects are frequent. Coarse grain locks avoid this programming complexity, but also come with limited scalability and, thus, lack of performance at high thread counts.

Here TM promises the simplicity of coarse grain locks with the performance of fine grain locks. Moreover, examples show that composing two operations from two separate code pieces under locks is not always possible [117]. Especially if the code must maintain certain invariants, the programmer has to redesign and rewrite these operations in one combined operation. TM enables this composability of rather simple operations through strong guarantees associated with transactions [3]. As an example, Adl-Tabatabai et al. demonstrate how to realize a move operation between two concurrent hashtables with transactions. This solution is as simple to write as with a coarse grain lock and maintains the invariant that the moving element must be present in only one hashtable at a time. With fine grain locks, the programmer must modify the underlying concurrent hashtables to implement a move operation and may easily introduce deadlocks. With transactions, the programmer only has to wrap the move operation in a transaction. The properties of transactions, which originate from the database community, will assure the desired behavior. The idea of transferring the properties from the database community to the synchronization (of processes) dates back to 1977 and has been proposed by Lomet [126]. Lomet also discovers that strong guarantees, e.g., the atomic actions, simplify the arguing of programmers about the correctness of code.

The properties that provide the strong guarantees for TM are atomicity, consistency, and isolation. Atomicity guarantees that either all operations in a transaction are performed or none of them. Consistency demands that no inconsistent states are introduced through transactional execution. Isolation assures that two concurrently running transactions appear to be isolated from each other. Thus, the computed results are independent of the intermediate state observed by a second transaction. TM can have a huge impact on large scale software development by enabling the composability of separately developed and maintained parallel software modules. Due to the strong guarantees and the high-level abstraction, combining parallel modules that synchronize with TM seems feasible.

```
__transaction {                __transaction {
  account2 -= value1;            account1 -= value2
  account1 += value1;            account3 += value2;
}                              }
```

Listing 2.1: Example of two threads executing transactions.

From a programmer's perspective the synchronization is severely simplified compared with fine-grained locking. Listing 2.1 illustrates two threads that are concurrently executing transactions. The block following the keyword `__transaction` is executed with ACI properties. In this example the account variables are located in shared memory and the value variables in local memory (e.g., as stack variables). Thus, there is a dependency between transaction 1 and 2 because of the variable `account1`. In case both transactions are executed in parallel, this dependency will likely lead to a conflict that needs to be

detected and resolved by the TM system. A conflict is caused by two transactional accesses from two threads to the same address where at least one is a write access. The resolution of a conflict mostly requires to abort one transaction, revert its changes, and retry the transaction.

2.2 Realization of Transactional Memory

In order to realize a TM run time system that provides the ACI properties, the TM system solves the following problems. An intermediate value that a transaction produces before the commit must be shielded from accesses of other transactional accesses because it is still speculative. Hence, a conflict with another transaction, that is caused by two accesses to the same address where at least one is a write access, may still abort the ongoing transaction and revert the speculative values. These conflicts may be detected when accessing the address or at commit time of the transaction. Another aspect that needs to be addressed is the versioning of data. A speculative value is associated with a version number, also called data versioning, so that the TM system may perform a validation that detects reads of outdated values and aborts the transaction during the commit phase. This is required to assure the serializability of transactions.

Hence, a TM system that implements these techniques makes important design decisions regarding the conflict detection and the buffering of speculative values that influence the performance. A conflict detection scheme can either be optimistic or pessimistic. Optimistic conflict detection checks for conflicts at commit time whereas a pessimistic scheme continuously checks for conflicts at access time. All reads and writes are monitored in so called read/write sets of a transaction. These serve as meta data for the TM algorithm that detects conflicts and enables a roll back of a transaction. A similar design decision has to be made for the buffering of speculative values. The optimistic scheme updates the memory in-place such that an undo log must be maintained by the TM run time system in order to enable a rollback. The pessimistic scheme uses a write buffer to temporarily buffer speculative values.

Another important property of a TM system is the resolution of the memory accesses inside the TM system. This resolution defines the granularity for the conflict detection. Depending on the source language and the realization of the TM run time system the following granularities are wide-spread: object-based, word-based, stripe-based, or cache line-based conflict detection. Object oriented languages feature an object-based conflict detection granularity, that also comes with the advantage of storing TM meta data with the objects which increases the data locality and employs a special open operation prior to performing transactional operations on the object. A stripe-based conflict detection scheme divides the memory space into stripes of fixed size that form the granularity for detecting conflicts. The word-based and cache line-based schemes are special cases of the stripe-based scheme that support resolution on the size of words and cache lines respectively. These conflict detection schemes are employed in languages without run time system or object-orientation, e.g., `C`. The word-based scheme requires more memory for meta data whereas the cache line-based resolution may exhibit *false positives*. These are accesses that are falsely classified as conflicts due to the granularity of the resolution. These accesses go to distinct memory location but map to the same STM internal lock. With a higher resolution of the conflict detection scheme, these accesses would not be classified as conflicts. This problem is analogous to the false sharing problem. We explain both problems in detail and address the problem of false positives in Section 7.2.

When two transactions execute concurrently both may access the same memory address. In case at least one of these accesses is write access, the two transactions have a conflict. This situation is called contention because both transactions contend for the same address. The TM system provides means for contention management that decide which of the transactions should abort and retry. These contention management schemes may assure e.g., fairness so that each transaction is allowed to commit at some point or progress of the application. Contention management schemes often employ a delay time before the transaction is allowed to retry. The delay avoids that the two transactions immediately conflict again. Often the delay is determined through an exponential back-off mechanism so that the delay increases exponentially each time the transaction retries again.

These concepts can be realized in software as Software Transactional Memory (STM), hardware as Hardware Transactional Memory (HTM), or as a combination of both as hybrid TM. In the following, a few representatives of the respective species will be presented in this thesis.

The interested reader finds a more complete overview of TM systems in [83]. This overview is beyond the scope of this thesis so that we restrict our discussion to a few example systems that illustrate the wealth of possible implementation options of the required actions and design decisions in TM systems that influence the run time behavior. The inexperienced application developer faces all these TM systems with their possibilities. This overview enables the reader to understand the dilemma that the application developer faces and that we research to alleviate with tool support for TM that is presented in Chapter 4 and the following.

2.2.1 Software Transactional Memory

Software Transactional Memory (STM) implements the algorithms that maintain the ACI properties for the programmer in software. Due to the optimistic concurrency with TM, the STM system must buffer speculative data while transactions are in progress. Moreover, multiple versions of the same datum may exist when transactions execute concurrently. Thus, the STM requires a data versioning system to assure that only consistent states are committed to main memory. During the commit phase these versions are validated to assure that no outdated values produced the results. Therefore an STM systems implement algorithms for conflict detection and validation in software [194]. Moreover, STMs may buffer speculative data to perform data versioning in two ways: in-place that is the speculative version of the data is kept in memory and the STM maintains an undo log or in a write buffer so that speculative data is buffered inside the STM and made publicly available during a successful commit operation. The conflict detection can either be early/eager or late/lazy. These strategies are also known as encounter-time locking and commit-time locking strategies. Further, STMs support either visible or invisible read operations. Visible reads can be observed by concurrently running transactions, e.g., through acquiring a lock that corresponds to the address, whereas invisible reads can not be observed. Of course, implementing the selected algorithm and data versioning in software comes with significant overheads [27]. In the following, we discuss the commit operation in an STM system in detail to illustrate the basic concept of STMs.

According to Fraser and Harris [67], the commit in an STM system goes through three phases. Figure 2.1 illustrates these phases. In the example the transaction comprises a read to address a1 and a write to address a2 – both of these accesses are stored in the read and write set respectively. The commit operation starts, visualized by the gray bar, and first

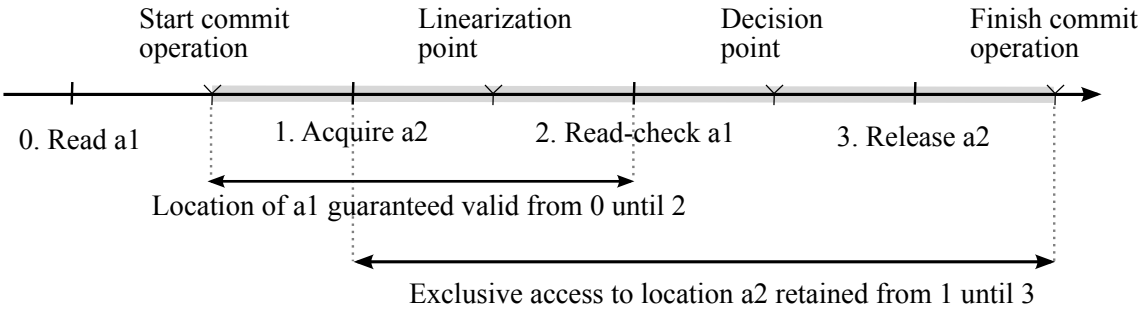


Figure 2.1: Illustration of commit phases in a Software Transactional Memory system; taken from Fraser and Harris [67].

acquires exclusive access to the locations that need to be written (in this case a2). After exclusive access is achieved, no other transaction can update these locations. Thus, this is the logical point in time where the results of the transaction take effect with respect to other transactions. This point in time is also called the linearization point because if this transaction commits successfully (which is undecided yet), this point determines the rank of the transaction in the serialization order with respect to other transactions. Although the transactions potentially execute at the same time, this determines the order in which changes to the contents of the memory are made. This guarantees that a transaction with rank k will use the newly produced memory contents of transaction with rank $k-1$ if there are overlaps in the memory read and written. The next step in the commit operation assures this through validating the reads of the transaction. In case a read of an outdated variable is detected, the transaction rolls back. This means that a different transaction modified the variable contents between the read and the read check. The read check relies on versioning information and is also called validation phase. The decision point is the point in time where the decision whether the transaction commits or aborts is made. In case of a commit, the variables that are written are updated with the new values. Regardless whether the transaction commits or aborts, it needs to revoke the previously acquired exclusive access to the variables. This step completes the commit operation.

The first pure software implementation of a TM system originated in 1995 [180]. This approach is known as a static solution because the programmer needed to identify all accesses to shared memory locations in a transaction and pass them with the function call at the begin of the transaction. This procedure was error-prone and hindered straightforward refinement of transactions because programmers could simply forget to add a new location to the list of accessed locations.

Thus, later STM versions support accessing memory locations dynamically and are called Dynamic Software Transactional Memory (*DSTM*). DSTM has been proposed by Herlihy et al. [90]. DSTM uses a write buffer and conflict detection at the granularity of objects.

Harris and Fraser present a Word-based Software Transactional Memory (*WSTM*) and follow a similar approach using a write buffering STM but with conflict detection at word-size [67].

Dice et al. published a popular TM algorithm, called Transactional Locking II (*TL2*), in 2006 [54]. The algorithm uses a global counter, called *global version clock*, to track the version of the values read and locks to detect writes to memory locations. Writing transactions increment the global version clock upon commit. A transaction samples the counter and stores it in a thread local variable. The algorithm stores reads in a read set

that contains the address read and writes in a write set that contains the address and the new value. Reads additionally check the corresponding write lock of the address to detect concurrent writers and abort the transaction early. The algorithm first locks the location in the write set on commit, then atomically increments the global version clock. In case the not all locations of the write set were acquired, the transaction fails. Then, the algorithm validates the read set and aborts the transaction when it detects a newer write to a memory location. On success, the algorithm commits the changes to memory and releases the locks on the locations written. The global version clock becomes a scalability bottleneck for high thread counts. Therefore, *SkySTM*, the successor of TL2, introduces the *scalable non-zero indicator*, a hierarchical data structure that replaces the global version clock, but provides the same functionality and is designed for high thread counts to improve the scalability of the implementation [124].

TinySTM[1] detects conflicts on a word granularity and comes with a customizable design. At compile time the programmer may select whether write-buffering or in-place updates are preferable and decide between a encounter-time or commit-time locking strategy [62]. TinySTM uses a similar conflict detection algorithm like TL2 but incrementally constructs a valid snapshot of memory locations for read-only transactions so that these do not need to be validated on commit. This approach is called lazy snapshot algorithm [162]. The linearization point of these read-only transaction is at the start time of the transaction (not during the commit phase).

Intel's *McRT* [168] comes with compiler support and an STM. The STM supports conflict detection at cache line and object granularity and implements an undo log as well as a write buffer.

A major difference in the implementation and design of e.g., WSTM and TinySTM is that WSTM is a lock-free (or more precisely non-blocking) implementation whereas TinySTM uses locks. In a lock-based implementation a transaction may acquire a lock and yields exclusive ownership of a memory location so that other transactions that also want to access this location have to block until the lock becomes available or have to retry the transaction. The conflict detection algorithm in a lock-free STM relies on architectural support for atomic operations e.g., compare-and-swap (not locks) so that it does not block other transactions. The emerging lock-free STMs are inspired by lock-free implementations of parallel data structures e.g., double-ended queues [88]. The data structures synchronize concurrent accesses to a parallel data structure, in this case a queue that supports adding and removing elements at both ends, by means of synchronization primitives that give weaker guarantees than locks and the use of specifically designed algorithms. In this case the correct synchronization of concurrent accesses is achieved through the use of atomic compare-and-swap operations. The atomic compare-and-swap operations can also be used to implement locks.

These non-blocking algorithms are classified according to the following scheme that groups them according to their *progress guarantees* [67]. *Obstruction-freedom* guarantees progress only if operations of one thread do not contend with operations of other threads. In a *lock-free* system, obstruction freedom is enhanced by the guarantee that – even in the presence of contention with other threads – the system makes progress. A common solution adds a helper mechanism. In case two threads contend for a memory location one thread helps the second thread to complete the ongoing transaction. This helper mechanism

[1] `http://tinystm.org/tinystm`

guarantees progress even in the presence of threads that contend for the same memory locations. *Wait-freedom* is achieved if all threads make progress – even in the presence of contention.

In [67], Fraser et al. present a word-based STM design (*WSTM*), an object-based STM design (*OSTM*), and a multiword compare-and-swap (*MCAS*) operation. The performance comparison uses these designs and compares them to lock-based and compare-and-swap-based synchronization to implement a red black tree, that is a self-balancing tree, and a skip list. The results show that the scalability of lock-based synchronization depends on the granularity of the lock on the one hand and the contention of the lock on the other hand. Also under high contention, the OSTM and WSTM designs outperform the lock-based synchronization.

Opposing to Fraser's work on obstruction-free STMs, Ennals designs an object-based STM that is not obstruction-free [58]. Instead he argues that obstruction-freedom is required for distributed systems, but is dispensable for STMs. Programmers appreciate the advantages of STM over serialized execution and do not need a guarantee that one atomic operation does not block another one. Furthermore the operating system/TM run time may adjust the number of concurrently running transactions so that transactions are not often suspended while running. Hence, transactions are suspended infrequently so that these transactions do not block other transactions which is a major concern of obstruction-free approaches. Ennals states that obstruction-freedom prevents important optimizations such as storing STM meta data with the object or in a private memory region of the current processor so that extra cache misses are unlikely. Moreover, choosing the number of active transactions smaller or equal to the number of cores should avoid conflicts. The STM builds on a revocable two phase locking scheme for writes (similar to TL2) and optimistic concurrency control for reads, both using a global clock for versioning. The memory is divided in a private space for meta data and public (shared) memory. A fair comparison with Fraser's STM, using the exact same setup and benchmarks, shows that Ennal's design performs between a factor of 2 (with limited memory bandwidth) and 5 (with high contention) faster than Fraser's. These results illustrate that a non-obstruction-free STM can perform significantly better than an obstruction-free one.

Another aspect of Software Transactional Memory systems is the contention management. The *contention manager* arbitrates in case of a conflict between two transactions. In most cases one of the transactions must retry whereas the other transaction continues. A contention manager may provide desired properties such as forward progress or fairness. Therefore, the contention manager must select the right transaction. A smart strategy also avoids livelock and starvation. Spear et al. present an example of a contention manager that is designed for blocking STMs with invisible reads that ensures fairness [192]. This algorithm reduces the accesses to meta data on every transactional memory access which has been one of the weaknesses of previous contention management strategies [170]. As a side note it should be noted that STM conflict detection policies such as commit-time locking or encounter-time locking also influence the contention management. The reason is that a conflict is detected earlierer with encounter-time locking providing fewer information about the transaction to the contention manager. With commit-time locking, the contention manager may decide based on more information e.g., read set/write set sizes that influence the rollback costs. Advanced techniques have been developed to benchmark contention management schemes: Discrete event simulation is used as a tool for the development of TM contention managers [52].

Contention management can either be *passive* or *active* [192]. *Passive* signifies that a transaction aborts itself if it detects a conflict with a concurrent writer. Applications with a regular access pattern benefit from this strategy. Whereas *active* would abort the concurrent transaction in case a transaction detects a conflict. In this thesis, we restrict the considered contention management strategy with STM to be *passive* due to the applications with regular accesses that benefit from this scheme.

An additional issue arises with the optimistic concurrency of TM when a transaction has to execute actions that can not be undone. This is the case with performing transactional I/O or more generally executing system calls from inside of a transaction. In the literature two terms exist to descibe techniques that enable to execute these actions: *irrevocability* and *inevitability*. Welc et al. research how to apply irrevocable transactions to speedup contention management and discuss the performance of irrevocability in general [206]. This approach introduces the *single owner read lock* that allows one transaction at a time to transition to irrevocable mode. Thus, one irrevocable transaction executes in parallel with revocable transactions. Spear et al. solve the same problem by introducing the inevitable mode and research ways of implementing and exploiting it [195].

This paragraph gives a brief insight into the evolution and debate of STM research and demonstrates that an application developer faces a variety of questions that require answering before choosing a fitting STM system. In their search for STM systems with different progress guarantees, contention management schemes, conflict detection algorithms, and the like, researchers have lost sight of the inexperienced application developer who has been the target audience for the invention of TM. In this thesis, we present techniques that make the programmer aware of the TM application's behavior and bridge the gap between the experienced TM researcher and inexperienced application developer by providing tools that help both groups to optimize the TM application.

2.2.2 Hardware Transactional Memory

In a Hardware Transactional Memory the TM functionality is implemented in hardware. Herlihy and Moss proposed such a TM implementation in 1993 for a (bus-based) multiprocessor architecture with shared memory [91]. The TM functionality is provided through extensions of the cache coherency protocol and adding a fully-associative transactional cache of fixed size in addition to the regular cache. Both caches are L1 data caches. All transactional operations, that comprise data loaded or stored in a transaction, target the transactional cache. Data can only be in one of both caches at a time. The transactional cache buffers the speculative values while transactions are in flight and supports the commit and abort of transactions. Moreover, the cache-coherency protocol is extended with transactional read operations. These may retrieve values from the normal caches or from the transactional caches if the tag of the cache line indicates a non-transactional value. Transactional cache lines are not transferred via the cache-coherency protocol while a transaction is in progress. This data is made available to other processors only after a successful commit operation that changes the state of the cache lines in the transactional caches to valid. Transactions that access only a certain number of memory locations so that the reads and writes fit in the transactional cache and do not exceed the scheduling quantum can be executed. Larger or longer running transactions fail and retry or will require software emulation. This first HTM proposal is an example for a *best-effort* HTM system. For the transactions that fail or retry due to resource limitations, there is no additional effort put in lifting the restrictions of the hardware. Many other implementation

alternatives have been proposed since then to lift these restrictions for hardware TM. A comprehensive description of HTM components is published in [93]. A short, incomplete survey of HTMs follows.

Transactional Coherency and Consistency (*TCC*), developed at Stanford University, has been proposed as a shared memory model [80]. TCC uses transactions as units for memory coherency and consistency which means that all writes of a successfully committed transaction are broadcasted to all participating processors. This assures the consistency because all processors receive the new values to work with and implements the coherency through the broadcast operation. The granularity of coherency and consistency is the transaction that has been defined through the programmer. TCC further uses a write buffer to store speculative writes and a commit control unit that grants the right to commit and ensures serialization at commits. Moreover, the commit unit also enforces a programmer-defined order of committing transactions that is specific to TCC. *ATLAS* is an implementation of TCC on FPGAs that uses a write buffer and optimistic conflict detection [147].

The following HTM systems use a pessimistic conflict detection and a write buffer: *LTM* (Large TM) developed at the Massachusetts Institute of Technology [6], Herlihy/Moss TM [91], and *VTM* (Virtual TM) [158] proposed by Intel Corp. and Brown University. An undo log and pessimistic conflict detection is used by the following HTMs: *UTM* (Unbounded TM) Massachusetts Institute of Technology [6] and *LogTM* (Log-based TM) proposed by the University of Wisconsin-Madison [138, 212]. In both cases a transaction may exceed the size of the cache and both HTMs provide sophisticated mechanisms to assure the data consistency and track overflowed values. These measures are indicative for unbounded HTMs, that lift the restrictions of the HTM system, and the accompanying complexity make them less likely to be adapted by industry. Issues with HTM are the occurrence of interrupts inside of a transaction or reaching the end of a time slice while executing a transaction. The common HTM behavior is to abort the transaction in these cases. To overcome these deficiencies and avoid an overly complicated HTM design, researchers proposed to combine hardware and software to a hybrid TM system.

IBM's *Blue Gene/Q* architecture supports Transactional Memory in hardware through buffering speculative values in the L2 cache and performing conflict detection in hardware [201]. The TM run time system configures the hardware at the begin of a transaction so that conflicting accesses during the execution of a transaction can be detected in hardware when accessing the L2 cache. The TM run time and the operating system monitor the execution of a transaction and provide a sandboxed execution environment called *jail mode*. This environment enables the execution of arbitrary code inside transactions. Especially the operating system plays an important role because it detects and resolves violations of the jail mode. E.g., I/O operations violate the jail mode because they can not be undone easily. Hence, these operations lead to a retry of the transaction in an irrevocable mode. The interplay of OS and TM run time system enables the execution of arbitrary code and guarantees forward progress through the irrevocable mode. Moreover, the run time system also adapts the amount of retries a transaction makes before transitioning into the irrevocable mode. A compiler/run time approach stores and restores the live-in registers. These operations cause some overhead. To classify the overheads, the paper compares BG/Q hardware TM with two different execution modes with TinySTM and `omp critical`. A comparison of the execution times of a single-thread with a sequential version gives first insights in the overheads and more results show the impact on accesses to the L1 cache and the length of the instruction path. STAMP benchmarks [24] illustrate the scalability when

going from 1 to 64 threads. Four different categories describe the relation between HTM and STM. *Effective HTM* comprises the benchmarks `genome` and `vacation`. Both show a better scalability than TinySTM. Benchmarks `ssca2` and `labyrinth` fall into the category *capacity bottleneck* because both exceed HTM resources. The TinySTM code for `labyrinth` uses privatization (cf. Section 2.3) and performs better. The same holds for benchmark `bayes`. For the other benchmarks of the STAMP suite, HTM shows the best performance at low thread counts, which is explained through limited hardware and coarse grain conflict detection that may become a bottleneck due to contention.

For a comprehensive description of the BG/Q architecture and comparisons with other hardware please confer Section 8.2 of this thesis. Currently the BG/Q architecture is the only commercially available architecture with hardware transactional memory support. In this thesis we research what application properties benefit from hardware transactional memory on BG/Q in Chapter 8 and highlight a profiling and a tracing tool that both account for transactional execution in Chapter 9. Other vendors also announced microprocessors with HTM support. Intel®'s next processor generation, called Haswell, features Transactional Synchronization Extensions (Intel® TSX) [103]. The techniques and tools researched in this thesis should be transferred to Intel® TSX in the future.

2.2.3 Hybrid Transactional Memory

Hybrid Transactional Memory combines the strengths of HTM and STM components. *HaSTM* [169] uses HTM primitives to accelerate software routines e.g., for the validation of the read and write sets. Thus, the STM part is always active but relies on special hardware to accelerate the processing. Other approaches combine hardware and software in a different way. Transactions have two code paths: one that utilizes the HTM ressources, e.g., transactional cache and special instructions, and one that addresses the STM system. A transaction will first try to execute the code path with the HTM support because the execution is usually faster and if HTM resources are exhausted and the transaction retries (potentially a pre-defined number of times), the implementation falls back to the execution of the software path [47]. The hybrid TM system keeps the HTM and the STM meta data coherent. This requires that the HTM system also accesses STM meta data so that both systems can run in parallel and detect conflicts. A different approach executes either the hardware code path or the software code path of all transactions at a time [122]. Transactions that can not complete in hardware can queue for execution in software. In case enough transactions are in the queue, the execution in software is triggered. This approach does not have to keep STM and HTM meta data coherent because only one TM system is active at a time.

The combination of hardware and software to a hybrid TM system has the advantage that hardware costs are modest and the TM extension only needs to provide *best-effort* hardware. In combination with the software part, the hybrid system still ensures the execution of transactions that exceed the limits of the hardware. Although this execution comes with a performance penalty compared with executing transactions in hardware only, this approach makes a TM extension of current microprocessors affordable for manufacturers and does not expose the limits of the hardware to the programmer.

AMD's Advanced Synchronization Facility (*ASF*) is such a best-effort hardware extension of the AMD64 architecture that provides progress guarantees for transactions in the absence of contention of different threads for the same memory locations [36]. The instruction set extension provides the following ASF-specfic instructions: `speculate`, `commit`, `lock`

`mov`, `watchr`, `watchw`, and `release`. While the first two instructions start and end a transactions respectively, the other instructions declare protected memory locations and define for which kind of accesses these should be monitored. The `release` instruction removes an address from the read or write sets so that it will not be monitored any longer. Two implementation alternatives of ASF exist: a cache-based implementation that modifies the cache-coherency protocol and a variant that adds a locked-line buffer to each CPU that holds monitored addresses and backup copies. The evalutation uses a near cycle-accurate simulator and integer set and STAMP benchmarks [24] with up to 8 threads and shows a significant improvement over the STM solution.

Another hybrid TM system that also features an ASF implementation is *TMbox* [190]. In this thesis, we present a tracing methodology that extracts fine-grained event traces of the hybrid TM system TMbox in Section 5.2. Similarly to the ASF proposal, the new BG/Q architecture could also be classified as a hybrid TM system according to our standards because a TM run time system implemented in software initiates and commits transactions [201]. However, we refer to the BG/Q as HTM system because the compiler generates only one execution path for a transaction whereas hybrid systems, except for HaSTM, usually have at least two paths.

2.3 Memory Model for Transactional Memory

In the following, we will review the state-of-the-art that targets the definition of the memory model with respect to transactional execution. In general, a memory model defines the order in which loads and stores must execute to produce a well-defined result for the programmer. With shared memory, the model also defines when parallel threads of execution see values produced by other threads. For TM, the memory model also needs to define the semantics for parallel transactional and non-transactional accesses to the same shared datum. The memory model provides the foundation for programming with TM and, hence, influences our work that researches methods and strategies for the optimization of TM applications in multiple ways. For instrumenting accesses to shared memory with the compiler, the memory model defines which accesses to instrument. This especially applies to accesses to shared memory outside of transactions. For the optimization strategies, the memory model influences whether specific programming patterns have the potential to alter the results of the execution. Thus, the memory model has a huge influence on our work and although we do not research the memory model itself, we discuss current trends with respect to the TM architectures used in our work.

Ever since the proposition of sequential consistency in 1979 by Leslie Lamport [116], the memory model has been a controversial topic. Memory models that give strong guarantees to the programmer must restrain the optimization possibilities of the compiler and the architecture. On the other hand, systems that perform aggressive reordering of memory accesses and compiler optimizations exhibit a program behavior that is difficult to reason about and predict.

In the Transactional Memory domain, the memory model discussion concentrates on the relationship between transactional and non-transactional accesses and whether a programmer needs to be aware of implementation-level details of the TM system. The opposite positions are a *strong* and a *weak isolation* model. Strong isolation provides isolation between transactional and non-transactional accesses to shared memory [16, 185]. Hence, strong isolation prevents non-transactional read accesses to observe intermediate states

of a transaction in progress and enforces an order of the transaction and any kind of non-transactional access in case both access the same datum.

In Software Transactional Memory there are two ways to achieve strong isolation. A pure software solution must instrument non-transactional accesses to shared data with STM functions, so called single access transactions, to detect concurrent accesses. In this case the strong isolation properties come with severe overheads that result in a loss of performance. In case hardware primitives support the conflict detection, these can also be used to enforce strong isolation.

Weakly isolated transactions do not give any guarantees for concurrent transactional and non-transactional accesses to the same data structures [133]. Hence, intermediate states of a transaction could be observed with a non-transactional read access. This introduces problems when programmers want to use well-known parallel programming patterns like *privatization* [193] or *publication* [133] because these move data between a shared and a private memory space and, hence, access the data from inside as well as from outside of a transaction. A transaction may privatize data through moving it from a shared to a private memory space in order to process it without the need for synchronization. In a publication pattern the transaction moves private data to a shared memory space inside of a transaction. These two patterns in combination with an in-place update implementation of the STM may introduce race conditions in an otherwise race-free program. Programming with these patterns seems natural to most programmers so that a stronger model is required. The most intuitive model that is based on weak isolation is *single global lock atomicity*. In this model transactions behave as if they were acquiring a single global lock on start of a transaction and releasing it on commit [134]. A quiescence mechanism assures that each transaction waits until it is safe to commit without introducing a race in a potential privatization pattern. This assures the correct ordering of the transactional access that privatizes the data with respect to other transactions that access the same memory location and could introduce spurious undo operations. The inverse problem is solved through publication safety. For the single lock semantics, the implementation of the STM also must not expose speculative state to non-speculative accesses (e.g., not use in-place updates) and maintain granular safety which means only undo operations at the same granularity that accesses have been performed with [185].

Menon et al. compare the single global lock atomicity in `Java` with stronger and weaker models and come to the conclusion that strong isolation as well as single global lock atomicity come with significant overheads [134]. While strong isolation suffers from large single thread overheads and shows a good scalability, single global lock atomicity has lower single thread overhead but scales not as good.

Luchangco argues that defining semantics for Transactional Memory in terms of locks would lead to a diverse set of incompatible implementations which would break the composability of parallel modules that the TM community has hoped for [128]. The existence of different and incompatible isolation levels in the database community serves as an example. Further, two examples showcase why the author thinks that lock-based semantics are non-intuitive.

Dalessandro et al. propose a model that is stronger than weak isolation and weaker than strong isolation: Transactional Data Race Freedom (*TDRF*) assures transactional sequential consistent semantics for data race-free programs [46]. On the one hand, this does not give semantics to programs with data races as strong isolation does. On the other hand, the

memory accesses inside the transaction are performed in program order, giving a stronger and more intuitive guarantee than strong isolation that does not adhere to the ordering.

Spear et al. propose multiple implementations to achieve privatization safety [193]. The two basic concepts are introducing blocking fences or enabling non-blocking privatization. While the first technique ensure privatization safety through waiting and thus blocks the committing thread, the second technique must inspect the meta data of all memory operations for a privatizing transaction and perform full validation of the read and write set. These techniques can be refined by programmer annotations so that only affected transactions pay the performance penalty [214]. Although the proposed annotation performs quiescence of transactions as default, it introduces the risk that the programmer forgets to remove an annotation when transaction was changed to privatize data. The potential result is that the execution either fails or produces the wrong result.

Grossman et al. make a first attempt to explore the impact of high-level memory models on the semantic of transactions [79]. They research the impact of a weak memory model (weaker than sequential consistency) on compiler transformations and illustrate that the isolation as well as the ordering of transactions are affected by the memory model. Moore and Grossmann develop a language that is capable of modeling the effects of high-level transactions that execute with multiple small steps [137]. Along with multiple models for strong and weak isolation, this work also covers parallel nesting (e.g., spawning a thread in a transaction).

Harris et al. present a concurrency model that is based on transactional memory and enables composability [84]. They highlight the implementation of an STM in Haskell and show how transactions are separated from irrevocable actions such as performing input or output. Moreover they also present a formal definition of the operational semantics of STM Haskell.

Other researchers propose to enrich the transactional programming model with programmer-defined points that enable transactions to cooperate [188]. At these points transactions may observe speculative state of other transactions and communicate. This not only allows to call barriers inside transactions but also enables irreversible operations as well as having helper transactions. The model composes with the open nesting of transactions if compensating actions are supported. The programmer must specify the points when transactions may communicate and cooperate which complicates the programming model for the novice programmer.

Eventually the debate around a practical memory model for TM constructs in `C++` converges to a semantic that provides the guarantees of a single global lock that guards all transactions [18]. The semantic is expressed in terms of the memory model of `C++11` [5]. In practice the semantic means that a data access to a variable that is concurrently accessed outside and inside of a transaction is considered a data race and may yield a variety of legal results. Interestingly the memory model of the BG/Q architecture provides stronger guarantees [201]. A transactional execution of the case would be strongly isolated and force the transaction to roll back when the variable is accessed concurrently outside of a transaction. When the transaction executes in serial mode, multiple interleavings of the concurrent updates are possible. As a result, well-behaving software that is written with transactions following the specifications of the transactional constructs for `C++` should directly be transferable from STM to BG/Q. A transfer from BG/Q to STM might uncover data races that have been covered by the strong isolation of BG/Q. Thus, the portability of

TM software may cause some issues that could be alleviated using tools in the future. For the work presented in this thesis, the STM supports weak isolation while hybrid TM and BG/Q provide a stronger kind of isolation. For hybrid TM the execution in hardware mode is strongly isolated due to a non-transactional write sending an invalidation that aborts a transaction in flight. The same applies for BG/Q with the execution in the transactional mode. Thus, for executing in the transactional mode, optimizations that rely on patterns that require a stronger memory model, such as the privatization or publication patterns, are possible.

3. Related Work

This Chapter 3 reviews related work that is relevant to classify the work in this thesis with respect to other research. In order to facilitate the understanding of this works, we build on the knowledge about the concept of TM and the different implementation possibilities for TM as illustrated in the previous Chapter 2.

Section 3.1 gives an overview over related work that covers compiler support for TM. These works prepare the reader for our work presented in Chapter 7 that not only introduces basic TM support for GCC, but also explores how to use static information to select an STM property.

Then Section 3.2 discusses techniques for the retrieval of run time information of applications in general and more specifically for TM applications to establish a basis for the work on generating TM event traces in STMs presented in Section 5.1. Section 3.3 reviews tools for the optimization of TM applications and compares these approaches with our work presented in Chapter 6.

Section 3.4 presents research that focuses on hybrid TM systems, profiling of TM and other applications, and highlights approaches that are based on FPGA hardware. These works are the fundament for the extension of an FPGA-based hybrid TM system with event logging extensions in Section 5.2.

In Section 3.5, we highlight case studies that document how to program with TM. First, we will highlight studies that involve student groups. Then, we review the applicability of TM to scientific codes and other application areas. From our perspective, applying the tools researched in this thesis would have been beneficial for all of these works.

Section 3.6 holds works that cover the modeling of performance with TM and the energy consumption of TM system. Although these approaches are not directly related to the methods and strategies researched in this thesis, the presentation of works from nearby research areas helps to generate new ideas for future work.

Then, Section 3.7 presents approaches that cover the adaptation of the STM system. These approaches are orthogonal to the ones presented in this thesis and could, thus, be combined.

Finally, Section 3.8 reviews related work that targets the phase detection in sequential and parallel programs. This provides the ground for the phase detection algorithms for TM applications proposed and evaluated in Section 6.4.

3.1 Compiler Support for TM

In this section we will present related work that targets compiler support for TM. The detailed presentation of the works lays the foundations to understand the novelty and the contributions of the two TM-centric compiler-based approaches in Chapter 7.

For a basic support of Transactional Memory an API-based approach is sufficient. Dalessandro et al. demonstrate that a well-designed API for `C++` can exercise language features such as macros, overloading, exceptions, and multiple inheritance to simplify the API, support privatization, and ensure usability through catching programming errors [45]. More importantly, they also state that this approach fails to deliver the ease of programming. Therefore, and for the improving of the programming model and providing optimizations the compiler plays the primary role.

A TM-enabled compiler additionally recognizes a keyword or pragma that marks a transaction in the code. Inside these transactions, the compiler substitutes shared memory accesses with calls to a Software Transactional Memory library (STM). Furthermore, live-in stack variables need to be logged on entry to the transaction so that these can be replayed when the transaction retries. For STM implementations without hardware support to accelerate conflict detection, executing these STM functions for reads and writes, also denoted as *barriers* in the following, has a large overhead. Thus, compiler optimizations that mitigate these overheads are of great importance.

Intel develops McRT, a runtime system for multicore architectures, which includes an STM library implementation and compiler techniques to optimize and support these STM barriers in an unmanaged language [202]. This work transfers previous algorithms for conflict detection for `Java` [4] to unmanaged languages e.g., `C` and `C++`. Further, Wang et al. present solutions to handle function pointers, the aliasing of local variables, and exceptions introduced through inconsistencies caused by speculation [202]. Compiler optimizations perform redundant barrier elimination, inlining of the STM fast path, elimination of barriers that refer to transaction-local memory, and generating different functions for transactional and non-transactional execution. Further, a TM-specific register checkpointing scheme enables partial redundancy elimination across transaction boundaries. For the *SPLASH-2* benchmarks [210] the single-thread overhead of STM compared with fine-grained locking (both with optimizations) is $6.4\,\%$ on average. On a 16-way SMP system executing 16 threads, the benchmark `raytrace` scales best with STM whereas the benchmark `barnes` does not perform as good as with fine grain locks. Both perform always better than coarse grain locks.

In addition to compiler support for transactions, McRT also provides transactional versions of `C` library functions such as `malloc` and `free` [97]. The implementation of these uses thread-local storage to avoid introducing atomic primitives for the synchronization of threads. Further, memory allocated in aborting transactions is freed correctly and nested transactions with partial rollbacks are also supported. Integrating `malloc` and `free` with the STM enables to recycle memory (which leads to an increased locality) and use object meta-data for conflict detection. Most importantly this integration enables the programmer to manage memory from inside of transactions.

For managed languages such as `Java`, optimizations can also be implemented in a just-in-time compiler/runtime environment. This approach has the great advantage that an intermediate representation of the code can be tuned to the underlying platform. Adl-Tabatabai et al. research compiler and runtime support for STM in `Java` [4]. They apply

just-in-time optimizations to STM operations in order to safely eliminate redundant barriers and to reduce the single thread overhead. With these techniques the single thread overhead has been reduced from $63\,\%$ to $16\,\%$ with respect to lock-based synchronization. The TM system enables nested transactions with partial rollback to the checkpoint of the inner transaction and the conflict detection on a word or object granularity based on the type information.

Dragojević et al. propose a compiler and runtime approach for optimizing STM read and write barriers [56]. They observe that memory is often allocated inside of a transaction and later accessed in that or a subsequent transaction of the same thread through expensive STM read and write barriers. These barriers are unnecessary because the allocated memory has not escaped the transaction (or more precisely the thread) and may thus not be accessed by a different thread. They introduce the term *captured memory* for this kind of memory access. Moreover this approach also covers transactional accesses to read-only and thread-local data that can also benefit from eliding regular barriers.

A static approach for performing this so called *escape analysis* [34] requires inter- and intra-procedural analysis to cover all memory references. Since the compiler has to make conservative assumptions, it is complemented with a run time system that checks whether the address to be accessed has escaped previously. The implementation uses the standard pointer analysis in the Intel `C++` compiler. This analysis uses intra-procedural pointer analysis and function inlining to include function calls. The available inter-procedural analysis has not been used due to two reasons: it would add substantially to the compilation time and this approach would not work when using linked libraries. The performance of three data structures for implementing the address range checks in the run time are compared: tree, array, and filter. While tree always returns the exact result, array and filter may yield false negatives. The overall result is that leaving out the expensive barriers when accessing captured memory may yield up to $18\,\%$ performance improvement for 16 threads.

Wu et al. describe similar compiler optimizations for TM in `Java` and `C/C++` to identify and eliminate TM barriers for contention-free writes [211]. Escape analysis for heap locations and optimizations for stack-local variables, that have their address taken, are combined to reduce the amount of read and write barriers. For an implementation of a B+tree [121], that connects the nodes on one level with a single-linked list, applying both techniques together yields performance improvements of $32\,\%$ on average across 1 to 16 threads. A SQL relational database, HSQLDB[1] that holds a table in memory, serves as show case for the potential of TM. The existing multi-threaded version uses the `synchronized` statement to guard the database object. A transaction straightforwardly replaces this block. Although the single-thread performance is 5 times slower with TM, the better scalability of TM allows TM to overtake the lock-based version at 4 threads. Moreover, compiler optimizations (separate transactional version of functions and transaction local optimizations) are effective and increase the throughput of the TM system.

Some researchers propose transactional memory as an enhancement to OpenMP [136, 9]. These proposals include a variety of new transactional directives, such as `#pragma omp sections transaction` grouping together independent sections that are treated as transactions. *OpenTM* [9] extends GCC with two nesting variants: *open* and *closed* nesting. Open nesting publishes the state of an inner transaction in case the outer transaction aborts

[1] HSQLDB - $100\,\%$ `Java` Database available at `http://hsqldb.org/`.

whereas closed nesting discards the changes from the inner transactions causing no side effects; open nesting allows for additional optimizations to happen at compilation and runtime, but breaks major assumptions about transactional execution (it is intended for expert library developers). An extension to the `omp for` directive is also proposed, `omp transfor`, executing the loop iterations in parallel as transactions. Furthermore, the programmer may specify the scheduling of these loop iterations and enforce sequential commit of the transactions (relying on the quiescence mechanism) to enable a memory consistency behavior compatible with weakly isolated, single-lock execution [133]. Milovanović et al. [136] study the interaction between OpenMP 3.0 tasks and transactional execution. In particular, an optional list holds the shared memory locations to instrument or not instrument. This mechanism provides the programmer with a verbose yet effective means to reduce instrumentation overhead. A similar mechanism is proposed in IBM's TM-enabled *XL Compiler* [101].

From the previous paragraph, we conclude that both TM extensions of GCC are closely coupled with OpenMP. Thus, an extension with *Pthreads* or a comparable threading library other than OpenMP is not possible with these approaches. Moreover, these approaches also extend the OpenMP primitives so that they go beyond a transactional memory extension of GCC that is presented in Section 7.1.

Tanger is an open source compiler framework that supports the use of transactions [61]. This particular approach extends the Low Level Virtual Machine (*LLVM*) intermediate representation and generates code for the TinySTM library [62]. Further enhancements to Tanger allow the conflict detection algorithm of TinySTM to operate on objects in an unmanaged environment [161]. LLVM has been proposed in [118] and features a load-store architecture with an infinite register set. LLVM simplifies program analysis and optimization [119]. The compiler framework is very versatile and can also be used for worst case execution time analysis [148]. The universal applicability makes LLVM suited to implement a static analysis of the memory access patterns in transactions (cf. Section 7.2).

STM systems with conflict detection on cache line granularity are sensitive to *false sharing*. In cache-coherent multiprocessor systems false sharing may lead to cache *thrashing*. When different processors update different words of the same cache line in an interleaved manner, an invalidate-based coherence protocol makes the cache line oscillate between caches. The overhead associated with invalidating and transferring the cache line results in loss of performance. As false sharing is a major topic in Section 7.2, we briefly review related work on the detection and avoidance of false sharing. Tool support, such as Intel's *VTune Performance Analyzer*, helps to detect false sharing in general [102]. The programmer can also revert to manual techniques (e. g. employing the attribute `__declspec(align(n))` to enforce the alignment of data structures) or reorganize data structures so that accesses are not mapped in the same cache line. Further, programming patterns e.g., *privatization* and *publication* patterns help the programmer to circumvent false sharing. Privatization is based on copying shared data to thread local memory. Then the threads may process the data without interference of other threads, thus avoiding false sharing. Upon finishing the computation, the thread may copy the data back to shared memory, and herewith make the data publicly available. Menon et al. study how to employ these software patterns with transactions and their implications on the semantics of TM in [133]. The implications of supporting these particular programming patterns have been discussed in the previous Section 2.3.

The description of the false sharing pattern in the context of transactional memory, not only underlines the importance of tackling this issue but also promises potential performance gains. Hence, the static detection of a false sharing pattern in transactional memory accesses is a reasonable research direction. In Section 7.2 we present a heuristic that analyzes the memory access patterns in transactions, based on the results the compiler decides between a word-based and a cache line-based conflict detection scheme in the STM. Hence, our approach avoids the negative effects of false positives in STMs without relying on manual work or dynamic techniques.

In the following we review works that qualify and quantify the overheads of execution with STM. Yoo et al. identify four challenges that arise when running large-scale workloads such as a particle dynamics simulation and a game physics engine on the McRT-STM with Intel's compiler, supporting TM in `C` and `C++` [214]:

1. false conflicts that result from a coarse conflict detection granularity,
2. over instrumentation through instrumenting thread-local or non-escaped memory locations;
3. costs due to privatization safety that are caused by quiescing transactions,
4. poor amortization of STM overheads due to the lack of parallelism.

Caşcaval et al. research the reasons why Software Transactional Memory is limited to research approaches [27]. Their first observation is that no large-scale codes use STM although TM promises to improve the productivity and to ease the maintenance. Instead TM systems execute dedicated TM benchmark suites to find bottlenecks. They compare three STMs: TL2 and *IBM STM* with hand instrumentation with Intel's compiler/run time approach. The results with `delaunay`, `kmeans`, `vacation`, and `genome` (all from the STAMP suite [24]) show that on a quad-core Intel Xeon server mostly 4 threads are necessary to reach the performance of a single-threaded version. `Vacation`, executing with any of the STM variants, does not reach the single-thread performance even when executed with 8 threads. For IBM's STM a comparison shows the impact of compiler instrumentation versus hand instrumentation. Then the single thread overheads of two validation strategies show that a global clock is superior to full validation. Further for both global clock and full validation a simulator-based break down of time spent in STM functions verifies the initial finding. Further detailed break downs of the execution time of STM read, STM write, and STM end operations shows that the overhead associated with TM bookkeeping dominates the cost. With these findings, the conclusion states that there are great challenges ahead to make STM attractive. In order to keep the productivity gains, the programmer should not be involved (e.g., through annotations). In their opinion, these findings justify only a small investment in hardware support.

Common approaches for the language integration of Transactional Memory either extend the language with a keyword (or pragma) or provide a specific API so that the programmer can address the TM system. In the following a complimentary approach, called *Optimistic Thread Concurrency* is presented [75]. Azul Inc. addresses the large code base of legacy software that is written with lock-based primitives. Here, TM is used as an implementation technique to accelerate the execution of lock-based critical sections. In detail the `Java` virtual machine (Azul VM) is extended with mechanisms for speculatively releasing a lock and run on a hardware with TM extensions that track data contention. In case of contention, a rollback is initiated transparently for the application. The same technique

enables to optimistically execute synchronized blocks in `Java` and detect data contention. With profiling information the differentiating between different locking strategies becomes feasible. A conservative strategy, called *thick locking*, enforces mutual exclusion and is used in case of contention. *Speculative locks* lets more than one thread acquire this lock and checks for data contention of the memory accesses. In case of contention, one thread is aborted and resumes execution at the lock acquire operation without visible effects. *Thin locks* are used without contention and can revert either to speculative or to thick locks. For a hashtable implementation with $5\,\%$ writes, the optimistic thread concurrency shows a great improvement in operations per millisecond.

Welc et al. follow a similar idea to transparently reconcile locks and transactions [205]. Their scheme enables the monitor abstraction in `Java` to switch between locks and transactions. Locks are efficient for low contention cases because locks are optimized for this case through only setting a bit in the object using a compare-and-swap operation. In case of contention, locks would serialize the execution of threads, thus, transactions are the method of choice. However, there is an issue with the semantics of nesting in `Java` that needs to be resolved first. `Java` enables the arbitrary nesting of monitors. The results of a synchronized block are published at the exit of the monitor. When looking at closed nesting semantics of transactions, the inner transaction does not publish its state. To match both semantics, the authors identify atomicity idioms that the program must conform to so that the observable behavior is the same. A formal model, based on `ClassicJava` [66], shows that in the presence of that atomicity idioms both variants yield the same results. The implementation of the switching logic (based on the contention record of that monitor) is implemented in the Jives RVM. The *OO7* benchmark implements an object-oriented database management system [25]. A comparison of transactions-only with the hybrid scheme shows that the overhead in the uncontended case can be lowered and that speedups are possible with transactions in the contended case.

The last two works are not directly related with the work in this thesis, but have been presented to show that TM can also be applied as an implementation technique to speedup traditional synchronization approaches.

3.2 Information Retrieval in TM Systems

In this section, we will introduce the well-known techniques for retrieving information of a parallel application first and then review TM-specific solutions. There are several ways of retrieving information about the run time behavior of a parallel application. A very popular approach generates event traces at run time. Therefore the application must be instrumented with calls to event logging routines. This instrumentation can either be achieved statically at compile time or dynamically at run time.

Dynamic approaches often use dynamic binary instrumentation to instrument a running application. E.g., *Pin* is a dynamic compilation tool that features portable, transparent, and efficient instrumentation [129]. *DynamoRIO* belongs to the same category but differs in implementation details [22]. *Valgrind* is specialized on memory instructions and more heavy-weight [145].

Static instrumentation approaches often use libraries to log events. The *Open Trace Format* employs ASCII events and focuses on scalability and read performance [113]. More formats are the *Pablo Self-defining trace format* and *Pajé* [166, 50]. *Epilog* is a

representative for a binary trace format [207] and Intel's structured trace format (*STF*) for a proprietary trace format[2].

For the information retrieval in TM systems, the following solutions exist. A profiling solution for TM applications written in `C#` [220]. The profiling data consists of begin and start of a transaction with timestamps and transaction read and write set sizes. Conflicts trigger the recording of the winning transaction id and object. This approach takes advantage of the garbage collector so that additional information about objects is stored upon allocation and tracing tasks are executed after garbage collection in order to minimize the application disturbance. The garbage collector is characteristic for a *managed language*. In case of an *unmanaged language*, such as `C` or `C++`, different solutions are needed. For our approach (applications and STM written in `C`), more general solutions are needed.

Another approach implements a profiling framework for applications written in Haskell, thus, achieving a solution for a functional programming language [191]. Further related work uses discrete event simulation to support the development of TM contention managers [52]. Lourenço et al. propose a monitoring framework for TM with visualization capabilities that addresses the programming language `C/C++` [127]. The framework consists of four components: a monitoring tool and trace generator, a trace file processor, trace file analyzers, and a graphical user interface. We will discuss the monitoring tool with respect to our work here and the remaining components in Section 3.3. The monitoring unit reduces the throughput to $40\,\%$ of the original throughput (in transactions per second) for two test cases. The monitoring relies on event traces that comprise the same information as for our STM tracing but has been developed independently. Although they advertise their trace generation as having a low-overhead, the reduced throughput leaves some doubts on that claim. Compared with our work they also use thread-local buffers that hold tracing data in a binary format. Instead of having a fixed buffer that is flushed to disk when it is full, they merge all TM events according to their occurrence to establish a global order and write them to a single text trace file when the program finishes. This approach underestimates the amount of events generated when multiple threads continuously execute transactions for a few seconds: the system will either run out of memory because memory demands of the thread-local buffers increase linear in the run time or start swapping to fulfill the memory needs. In both cases the behavior of the application is altered through the tracing machinery. None of these approaches has explored techniques for compression nor did these works compare their performance with a binary-instrumentation tool; as we do in Section 5.1.

Castro et al. trace STM applications through intercepting the STM calls through the dynamic linking mechanism [29]. Their declared goal is application and STM library independence and low intrusiveness. They report a maximum overhead of $9.9\,\%$ when tracing start and commit events. This approach relies on the dynamic linking mechanisms of Linux. Their library define symbols with the same names as the STM library. Before executing the application, the tracing library is loaded into memory using the `LD_PRELOAD` mechanism. Thus, each call from the TM application calls the tracing library first. The trace library resolves the original symbol of the STM, traces the event in a per thread buffer and calls the STM library. Surprisingly a lock is used to protect the trace operation and the call to the STM library. The argumentation is that a call to the trace operation must be atomic because the order of events may break otherwise. This lock artificially serializes threads and, thus must affect the quality of the trace data. The authors do not comment on this

[2]Intel® Corp., Intel® Cluster Tools

issue. The experimental results section shows the number of commits and aborts plotted as absolute values over time for the STAMP benchmarks `genome`, `labyrinth` and `intruder` [24]. Castro et al. claim that this approach is independent of the STM library, this claim only holds for STM libraries with the same API or an API that has already been implemented in the trace library. Otherwise the interception of the calls in the STM run time fails due to the API of the STM being unknown to the trace library. Moreover, this approach works only with dynamically linked STMs. Often STMs are linked statically or STM functions may even be inlined by the compiler to reduce the overheads. In both of these cases, the presented approach fails. Due to relying on the interception mechanism, contents of STM internal data structures can not be traced and events can not be attributed to a transaction (only to a thread). Thus, our approach, illustrated in Section 5.1, allows to trace more information and, hence, provides information at a finer granularity to other tools in the post-processing step.

3.3 Tools for the Optimization of TM Applications

This paragraph holds the existing post-processing tools for TM applications. Starting with a generic debugging tool for TM, this paragraph also presents tools and frameworks for the visualization of the TM behavior that are comparable with our *VisOTMA* framework presented in Chapter 6.1.

Herlihy and Lev propose a generic debugging library for debugging TM programs that use STMs [87]. The design of the debugging library, called `tm_db`, is generic and modular so that it may interplay with different debuggers on one side and different STM libraries on the other side. Debugging tools for TM have to face the difficulty that intermediate states of the TM run time system should not be exposed to the user. Otherwise the debugging tool would expose details of the implementation that should be hidden under the covers of transactional properties such as atomicity and isolation. To achieve this goal, the authors define the logical value of memory location to be the value of the last committed transaction or a non-transactional access. The debugger presents this logical value to the user. In order to monitor the progress of a transaction, they define three different notions of transaction: the transaction as source code, the logical execution of a transaction, and potentially many (due to rollbacks) physical executions of a logical transaction. With this distinction, a user can track the progress of a transaction. Moreover, the transaction's status can be queried and distinguishing between covering and accessing a memory location enables to detect false conflicts. To debug problems with accesses at a sub-word granularity, a mask marks the accessed bytes. The user may also set breakpoints on transactional events and choose to observe events in different transaction-related scopes. The authors further show how they extended SkySTM [124] with a support layer and `tm_db` with a SkySTM specific module to eventually connect a debugger with SkySTM through `tm_db`.

Lev presents *T-PASS*, a transactional program analysis system, to profile transactional programs that use SkySTM [123]. T-PASS features an integration with the debugger and a technique for replaying transactions. The profiling tool helps to focus optimization efforts on frequently executed program parts. In order to tune these, Lev proposes the following strategies: split transactions to reduce contention, use privatization to reduce synchronization overheads, eliminate performance critical false conflicts and tune the configuration of the STM. The Dynamic Tracing framework (*DTrace*) [23] provides support for dynamically instrumenting software running in user or kernel space and has

been designed for the Solaris operating system. DTrace collects and aggregates events and passes them to T-PASS which complements these with sampling information. The overhead with T-PASS without profiling is $\approx 4\,\%$ and with profiling $\approx 10\,\%$. A single-threaded run provides information on how often a logical transaction is executed, how long it runs and the data it accesses. A multi-threaded run provides the cost of contention, data to analyze conflicts and data to identify writing transactional accesses that do not alter the object. Unfortunately, the description of the work does not cover which and how information is retrieved with that level of detail with the small overheads. The user interacts through queries with the `Java` front-end of T-PASS. Lev presents many diagrams that illustrate the capabilities of T-PASS. Again, a transaction has three abstraction levels (code block, logical and physical transaction) that are all supported through queries. Through distinguishing these levels, the user obtains a detailed view on the performance and may even correlate the data from different levels. The application under test is a hash table implementation with transactions. The charts show how to use filters and e.g., obtain a break down of unique locations accessed in a transaction divided up in read, writes and read-after-writes. T-PASS has a special emphasis on finding transactions with a small data set and short execution times that are candidates for acceleration through a best-effort HTM system [53]. A metric for measuring contention is defined as

$$contention\ overhead = \frac{failure\ time + wait\ time}{useful\ time}$$

where $failure\ time$ denotes time for retries including the times for exponential back-off due to contention, $wait\ time$ contains contention management and $useful\ time$ is for successful logical transactions. Further, a transaction may be inspected at the level of conflicts so that individual read and writes are classified into wins and losses. Wins indicate that this access caused a conflict but the other transaction aborted and losses are defined complementary. These metric assigns a weight to each memory access. With these weights, an optimization can focus on the accesses that have a higher impact on performance. T-PASS also supports the evaluation of the contention management strategy and identifying the upgrade of transactional reads to writes and writes that do not modify the object. T-PASS is implemented in `Java`, closely coupled with SkySTM and restricted to the use with the Solaris operating system due to using DTrace. The employed sampling techniques yield low overhead but selectively profile transactions. This delays the execution of the profiled transactions with respect to all other transactions. The randomized sampling mitigates the effect on the execution.

In the context of an object-based STM, profiling and in particular metrics to rate the application have been explored for `Java` [7]. The metrics comprise: speedup of the multi-threaded execution over a single threaded execution, execution time spent in transactions, wasted work defined as time in aborted transactions, mean aborts per commit, and percentage of time spent in contention management. Although these presented metrics are transferable to our approach, the implementation of a software profiling framework in `Java` differs significantly from our framework that addresses unmanaged languages and comes with a set of post-processing tools that includes phase detection in TM applications.

Moreover, a Haskell-specific profiling framework that profiles the TM application at the atomic block level has been proposed [191]. The advantage of this approach over the metrics that use averages in previous works is that it uncovers relationships between atomic blocks such as abort/commit due to conflicts and reveals the shared data that causes the conflicts. A global table contains the atomic blocks and the transactional variables accessed.

This table contains statistics on the number of aborted and committed transactions, the size of the read and write set, the cycles spent in the commit phase, work phase, validation and the number of conflicting variables. The overhead of the approach with an example with 20 atomic blocks using a linked list with more than 1000 conflicting transactional variables yields $7\,\%$ additional aborts on average. While this approach is demonstrated to have benefits with an implementation of the application and the run time system in Haskell, it is questionable whether this level of detail is practical to profile STMs written in `C/C++` due to a global table introducing an additional artificial point of synchronization. Thus, our solution favors a fine grained tracing approach combined with an analysis and visualization in a post-processing step.

Zyulkyarov et al. present a profiling and visualization framework for TM application's written in `C#` [220]. The profiling components are tightly integrated in the *Bartok-STM* library and rely on information of the garbage collector to correlate conflicting addresses with the source code location of the allocation. Processing of the profiling data is done offline or when the garbage collector is active (if this is possible) to reduce the probe effect on the application. A backtrace of the stack established the context of a conflict. All potential conflicts in a transaction are tracked through running all conflicting transactions to completion in serial mode and tracking all memory accesses. Profiling data records entry and exit of transactions with timestamps as well as read and write set sizes and gathers local and global statistics on the transactional execution. On conflicts, stack trace and conflicting addresses of the transactions involved as well as transaction id of the winning transaction are profiled. These are aggregated into *conflictWin* and *wastedWork* metrics that count how often a transactions has been winning a conflict or has been aborted. This information is turned into an aborts graph that represents the relationship between transactions. The visualization supports a timeline view that can be zoomed and helps to get an overview of the application's behavior. The paper presents three examples for the optimization of TM applications from the STAMP suite [24]. `Bayes` scaled poorly. The reason is the object-granularity of the Bartok-STM that generates false conflicts. `Intruder` comes with a red-black tree that caused many conflicts. Replacing it with a chaining hashtable reduces the conflicts and improves performance. Further, moving the modification of the queue data structure to the begin of the transaction reduced the amount of wasted work. `Labyrinth` has been modified applying application domain-specific knowledge and work with an outdated version of the data [203]. A key statement of this paper repeats a point also made in a previous publication [69] that states that two shorter transactions should be preferred over one large transaction. This environment has been tailored for an object-oriented language, in particular `C#` and a proprietary compiler and STM that performs conflict detection on object granularity. In this thesis we will transfer these techniques to an unmanaged language with an open source STM. Especially a suitable way of replacing the information available through the use of a garbage collector with means to retrieve the information in an unmanaged language is presented. Moreover, our approach additionally features the readings and correlation of transactions with hardware performance counters and researches techniques for hybrid TM and architectures with support for a proprietary hardware TM.

Recent related work in tools that visualize the TM performance also addresses the playback of execution traces [76]. These are visualized in soft real time which enables replay rates that range from 0.0001x to 1000x of the original execution. As opposed to previous work, this work enables the dynamic visualization of transactions and their cross correlations. Two recorded executions of the same program can be replayed side by side which reveals

differences in the execution through *TMProf*. TMProf buffers event traces internally and uses hashes to compress data structures of a transaction. This compressed representation of the traces is written to disk. TMProf's visualization consists of a static and a dynamic view to visualize the transaction's behavior. While the static view blends short and aborted transactions into a single large transaction, the dynamic view enables to distinguish these transactions through another color. TMProf is further used to improve and evaluate the previously best-performing contention manager, called `iFair CM`, of the *InvalSTM* [77]. TMProf reveals that `iFair CM` aborts readers in favor of writers. Changing this behavior so that single writers are aborted more often instead of multiple readers yields a new contention managers called `threadFair CM` that also accounts for the number of times a thread committed. `IBalanced CM` combines the strength of `iFair CM` and `threadFair CM`. These contention managers are evaluated with a program that executes a single writer and multiple readers in transactions, a transactional implementation of a red black tree, and a hashtable with transactions. The two new contention managers yield a higher throughput. TMProf is designed in a way that future TM hardware e.g., the *Haswell* processor, maps canonically to the already existing begin, abort, and commit events. This facilitates the integration of HTM events into TMProf in the future. In contrast to this work, our approach for STM also builds on compression and visualization with a static view. Our trace generation scheme does not take advantage of transaction-specific properties and is, hence, generally applicable to other use cases that benefit from generated event traces. Although our approach does not feature the replay of transactional events yet, our static view does not necessarily come with the issue that short aborted transactions appear as a single long transaction. On the one hand the programmer could use Paraver's zoom functionality for an in-depth inspection of the transactions. On the other hand, the programmer could click on one of the transactions and see the contained loads and stores so that the short nature of the transaction becomes obvious. These features allow the programmer to discover the length and behavior of transactions despite the static view. Moreover, other profiling and tracing solutions shown in this thesis already visualize the behavior of an existing HTM system - IBM's BG/Q architecture.

Lourenço et al. propose a monitoring framework for TM that addresses the programming language `C/C++` [127] with four components: a trace generator, a trace file processor, trace file analyzers, and a graphical user interface. Since the trace generator has already been discussed with respect to our work, we start with the trace file processor. This component serves as an event iterator that avoids loading the full trace file into memory. Further the trace file processor supports *savepoints* that mark a point in the trace file to enable jumping to that point. The trace file analyzers are divided into the chart type that they create. Statistical information is available through the use of `JFreeChart` and a timeline view through a newly developed `Java` Swing component. In this timeline view, a user may click on an aborted transaction to find the opponent transaction that will be highlighted with an arrow. Hence, compared with our work, not all dependencies between transactions are visualized so that only one abort reason is inspected at a time. This makes it hard to identify the transaction with the most dependencies in this view. The statistical information charts are comparable to our statistics but additionally provide statistics on the abort reason and accessed memory cells. Unfortunately, the framework does not describe a way to relate these memory accesses to the data structures of the application (as we do in our approach). Further, a mapping of the execution to the source code, that we already incorporate, is described as future work. The result section of the paper holds various charts of the statistics to compare a linked list and a red-black tree implementation with

TM. Instead of relying on a state-of-the-art visualization tool like Paraver, they chose to implement the visualization functionality in a new tool. Both approaches are valid and have pros and cons. Further, we demonstrate the versatility of the VisOTMA framework and additionally to the previous approach implement algorithms for the phase detection in TM applications (cf. Section 6.4).

3.4 FPGAs and Hybrid TM

This section presents related work from hybrid TM systems and profiling of TM and other applications with a realization in FPGA hardware. These works are related to the event logging extension of the TMbox architecture that we present in Section 5.2.

Field Programmable Gate Arrays (*FPGAs*) consist of programmable logic that can be used for the acceleration of full-system multiprocessor simulations [39, 40, 48, 49]. The flexibility of FPGAs is beneficial for architectural exploration. In this thesis, we exploit the FPGA properties to design and implement a low overhead monitoring infrastructure for a hybrid TM platform.

The tracing and profiling of non-TM programs has a long tradition as well as the search for the optimal profiling technique [13]. software techniques for profiling, targeting low overhead, have been researched [68, 144], alongside of Operating System support [216], and hardware support for profiling [51, 217]. Further, techniques to profile parallel programs using message passing communication have been developed [182]. In nearby research fields Faure, Benabdenbi and Pecheux describe an event-based distributed monitoring system for soft- and hardware malfunction detection [60].

Few works have been published in the context of studying Transactional Memory on FPGA prototypes. ATLAS is the first full-system prototype of an 8-way CMP system with PowerPC hard processor cores with TCC-like HTM support [204]. It features buffers for read and write sets and per-CPU caches that are augmented with transactional read and write bits. A ninth core runs Linux and serves operating system requests from other cores.

Related work also targets TM for embedded systems: Kachris and Kulkarni describe a basic and configurable TM implementation for embedded systems which can work without caches, using a central transactional controller on four Microblaze cores [108]. Pusceddu et al. present a single FPGA with support for Software Transactional Memory [157]. Recent work, that also utilizes MIPS soft cores, focuses on the design of the conflict detection mechanism that uses Bloom filters for an FPGA-based HTM [115]. Casper et al. propose the acceleration of transactional memory for commodity cores (*TMACC*) [28]. The conflict detection uses Bloom filters implemented on an FPGA. This accelerates the conflict detection of the STM. Moderate-length transactions benefit from the scheme whereas smaller transactions do not. TM support for beehive stores transactional data in a direct-mapped data cache and overflows to a victim buffer [38]. Bloom filters are also used for the conflict detection. Grinberg et al. demonstrate a system architecture to investigate Transactional Memory on Altera FPGAs [78]. NiosII soft cores are extended with engines for transaction dispatch, commit, and a rollback controller. A hierarchical fabric with configurable delays enables to mimic a distributed shared memory system that supports up to 16 processors. Statistic counters capture total cycles from program start to commit of the last transaction as well as average idle times due to ordering issues and other committing transfers.

Damron et al. present Hybrid Transactional Memory (*HyTM*) [47]. An approach that uses best-effort HTM to accelerate transactional execution. Transactions are attempted in HTM mode and retried in software. The HTM results are based on simulation.

Chung et al. gather statistics on the behavior of TM programs on hardware and describe the common case behavior [41]. Further, performance pathologies of hardware transactional memory are described and analyzed.

The programming model of the underlying hardware TM system of TMbox [190] is similar to the Transactional Coherence and Consistency model [80]. The monitoring techniques used in this work are also in some parts comparable to the Transactional Application Profiling Environment (*TAPE*) [31], both developed at Stanford University. Major differences include the use of multiple ring buses in the TMbox system, compared to a switched bus network with different timing characteristics and influences on HTM behavior. Further, the hardware support for profiling with TAPE incurs an average slowdown of $0.27\,\%$ and a maximum of $1.84\,\%$ [31] and does not take the profiling of software execution of transactions into account. Although the overhead looks small at first sight, it should be noted that even small changes in the timing of a thread may decide whether a dependency between transactions manifests as a conflict or not. Thus, based on this observation, the lowest possible profiling overhead is our goal.

Up to now, a comprehensive profiling environment for hybrid TM systems has not been proposed. Previous approaches either lack the ability to profile TM programs or are designed for a specific hardware or software TM system. As a consequence these approaches can not capture the application's behavior comprehensively. An application running on a hybrid TM system may transition between a hardware and a software execution mode. These changes can only be tracked and understood by a dedicated solution, such as the one presented in Section 5.2.

3.5 Programming with TM

In this section, we will review experience reports that describe the programming with TM in general. First, case studies with students describe the findings whether transactional memory is perceived to be easier to use. Then, expert programmers tackle the solving of scientific problems and other problems with TM. All of these approaches would benefit from using comprehensive methods and strategies for the optimization of TM applications and are, thus, potential future fields of application for the tools for TM developed in this thesis.

Case Studies with Students

Until now, the related work targeting the programmer of transactions is scarce. In a study conducted at the Universität Karlsruhe (TH), 12 students are randomly assigned to 6 groups [150]. All of these groups solve the same parallel programming assignment to implement a desktop search engine in `C` or `C++` and Pthreads. Three groups are allowed to use TM – in the form of STM with compiler support from Intel[3] – while the three remaining groups must use locks. Two instructors teach the students parallel programming, equip them with literature and continuously monitor the progress of the teams with semi-structured interviews [165]. This study follows the good practices described in [213, 215]. All

[3] Intel C++ STM compiler prototype edition 2.0.

teams complete the assignment although one TM teams needs to be excluded because their program crashes in the final benchmarking step. Another TM team wins this competition with the fastest execution time and had a faster development time than the best team using locks. These findings indicate that TM can increase the productivity when developing parallel codes and may even yield excellent run times. During the code review several cases for the misuse of the `abort` statement have been identified. In order to optimize the use of transactions, students implemented a broken pattern known as double-checked locking in the literature [8]. Further, a common mistake has been to read a shared variable without proper protection through a transaction while writing it outside of transactions. The authors concluded that performance as well as debugging tools are required to detect and resolve these cases. Another interesting point of this study is that TM performance has been found difficult to predict by the students which is backed up by the time spent for performance tuning. TM teams wrote small programs to benchmark TM performance. On the other hand, the TM teams spent less time on refactoring and debugging than the locks teams. A shorter and condensed version of this technical report has been published later [149].

Rossbach et al. [164] conducted a study at the University of Texas at Austin that covers the experience of 237 undergraduate students with parallel programming in `Java`. All students had to implement nine synchronization variants of a shooting gallery. Each of the following variants was to be implemented with fine-grain and coarse-grain locks, and TM. In the first variant multiple shooters (threads) would try to color one randomly selected lane in a color box. Only one lane access is allowed at a time. Only white lanes can be shot and when all lanes are colored, the last threads turns into a cleaner thread that colors all lanes with white. The second variant is similar to the first except that two lanes are colored in one attempt. The third variant demands a distinct cleaner thread that must be notified when the color box is colored. The study summarizes results from 3 semesters/years where the TM implementation changed from DSTM2 [89] to JDASTM [160] after the first year. These implementations had no compiler support, thus, students had to correctly place read and write barriers in transactions. This is a major difference compared to the previous study. As a consequence of the lack of compiler support, students regarded the required code as a barrier for transactional programming. The complexity of a coarse-grained lock and a transaction is rated equal. Moreover, both yield fewer errors than fine-grained locks. On average TM required more development time than coarse-grain locks but less than using fine-grained locks. The most remarkable finding is that error rates were lower when programming with TM.

Scientific codes

In contrast to the proposals of extending OpenMP with TM extensions [9, 136], Wong et al. state that it should be clear that TM is more than a research toy before extending the OpenMP standard [209]. The paper first introduces the IBM® XL C/C++ Enterprise Edition for AIX©Version 9.0 compiler and STM run time system [101]. The STM employs an optimistic scheme and detects conflicts on a 8 byte granularity. Further, function attributes as well as modifiers for the default behavior of the compiler are introduced. In case the programmer specifies a transactions with the attribute `notrans`, (s)he takes the responsibility for the proper instrumentation. Then, TM is applied to a scientific application, more precisely an unstructured-mesh multi-physics simulation, in order to judge the appropriateness of TM for this problem. The authors introduce the Benchmark for Unstructured-mesh Software Transactional Memory (*BUSTM*). This benchmark generates

conflicts in a random way. Built-in error detection facilities assure the correct execution. BUSTM is highly configurable: available cell types include: triangular prisms, hexahedra, tretrahedra, and pyramids. To resemble real unstructured mesh applications, BUSTM combines nodes, faces, and cells in almost arbitrary ways. This introduces the indirect indexing that is hard to synchronize efficiently with locks and the major point for employing TM. The results show that in a CFD-style experiment with deterministic conflicts the number of STM retries is lower than the number of errors detected in an unsynchronized setup. For a Monte Carlo-like experiment with probabilistic conflicts the opposite is true: many resolved STM conflicts and fewer errors in the unsynchronized setting. The number of conflicts increases with the number of threads for both experiments. As a concluding remark, the authors state that STM outperforms `omp critical` by 10%.

A second publication renews and strengthens the claim to include a special pragma to mark transactions, namely `#pragma tm_atomic`, in the OpenMP standard [15]. Instead of using the STM run time as in the previous study, this time the TM support is provided in the form of IBM Blue Gene/Q's Hardware Transactional Memory. For this the benchmark is renamed to Benchmark for UnStructured-mesh Transactional Memory (BUSTM). In this study performance is the primary focus. Thus, TM execution is compared with OpenMP atomic and critical. The experimental results show that for the prism mesh with an deterministic experiment the number of conflicts with HTM is comparable to the number of errors when running unsynchronized. Except for eight threads (no rollbacks), the number of rollbacks increases with the number of threads. TM performs better than `omp critical` for all threads counts and better than `omp atomic` up to eight threads. Then, `omp atomic` scales better up to 64 threads. For the tetrahedral mesh with an deterministic experiment, the number of rollbacks and errors shows the same trend for more than two threads. Again `omp atomic` scales linearly up to 16 threads, although TM is even 10% better up to eight threads. `Omp critical` is slowest for more than one thread. For the probabilistic experiments with the prism mesh, TM shows significantly more conflicts than errors reported by the unsynchronized version. Further, `omp atomic` and TM scale well, while `omp atomic` outperforms TM and shows a super linear speedup. For the tetrahedral mesh 16 threads generate the most conflicts although the most errors are reported for two threads. `Omp atomic` and TM scale well while TM again is not as fast as `omp atomic`. To summarize, TM can perform better than `omp atomic` if the transaction contains accesses to three memory locations. Further, the low conflict probability makes the presented algorithms well-suited for TM.

TM with Specific Application Areas

Bradel et al. investigate the applicability of Hardware Transactional Memory for trace-based parallelization of programs written in `Java` [20]. These `Java` programs investigated in this study exhibit recursions and stem from the *JOlden*[4] benchmark suite. These are `Java` programs designed for benchmarking the memory management system. The `C` benchmarks from Olden which are ported to `Java` are called JOlden. The HTM system used in this study is LogTM. For the benchmarks selected in this study, the approach of applying HTM yields an average speedup of 2.7 for four threads.

There are case studies that highlight the pitfalls when converting a parallelized application to use TM for all synchronization [219]. The application is a parallel implementation of the Quake game server. The transactions employed do system calls and I/O from inside of

[4] `ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz`

transactions, make use of nesting and may be short or long in length. Moreover, some data is also accessed transactional and non-transactional. A subsequent study compares two version of *QuakeTM*: a coarse-grained TM version with 8 transactions and a fine-grained TM version with 58 transactions [69]. The overhead with STM and the abort rate are the main factors that should be improved for a better performance with TM.

3.6 Performance, Energy, and Modeling of TM

This section holds works that cover the modeling of performance with TM, the energy consumption of TM system. Although these approaches are not directly related to the methods and strategies researched in this thesis, the presentation of works from nearby research areas may generate new ideas for future work. Thus, through this paragraphs future directions that could be explored by the methods and strategies for the optimization of TM applications.

In order to understand TM performance of a simulated HTM system, Porter and Witchel introduce *Syncchar* [156]. While previous related work reports the conflict behavior of HTM systems [186, 159, 151], this work develops a model and compares the prediction of the model against simulated execution times. Syncchar builds on two metrics: data independence and conflict density. Data independence represents the likelihood that threads will not access the same data. The formal definition for the data independence of a lock I_n is the mean number of threads not conflicting, provided n threads are concurrently executing critical sections protected by the same lock. The address sets of all threads are sampled and conflicts are detected through comparison of the sampled read and write sets. Sampling is applicable in this setup because the HTM system is simulated so that sampling the read or write set does not abort the running transaction. Threads involved in a conflict are recorded in C_n. Then, the data independence is computed as a running mean through $I_n = n - |C_n|$. The conflict density represents the length of the serial schedule that is necessary to resolve conflicting accesses. Thus, a high conflict density leads to a long serial schedule with many threads involved whereas a low density results in a short serial schedule. The formal definition is

$$D_n = \sum_{x \in C_n} \frac{\sum_{y \in \{C_n - y\}} conflicts(x, y)}{|C_n - x|}$$

with $conflicts(x, y)$ is 1 if the address sets of x and y conflict and 0 if not. These two metrics are the basis for estimating the performance of a lock-based code when turning the locks into transactions. Syncchar is evaluated with lock-based STAMP codes [24]. The error of this prediction compared with the transactional execution is 25% on average. The prediction of the tool deviates more if the run time of the benchmark becomes short with more threads (e.g., `yada` with 32 threads). Moreover, Syncchar has also been applied to optimize *TxLinux* [163], finding an unprotected writes outside and a read of the same address inside of a transaction. Once this issue has been fixed, a hang of bonnie++ is resolved and MAB shows a slight speedup while other program behavior is not influenced.

The analytical modeling of the performance of the TM conflict detection algorithms has also been covered [85]. Discrete-time Markov chains determine the algorithmic performance of the three different STM systems, enabling a rating of these systems. The dissimilarity of TM workloads has been researched with methods from statistics [100].

In the design process of multi-processor systems energy consumption plays an increasingly important role. This trend demands to research the mutual effects of combining Transactional Memory with techniques to reduce energy consumption. Hughes et al. apply dynamic voltage and frequency scaling and intelligent scheduling to optimize the throughput/power trade-off in HTMs [99]. Two policies are proposed and evaluated. First, dynamic voltage and frequency scaling is applied to times when a transaction stalls. Second, processing elements are clock gated when they execute an aborted transaction. Results are based on simulation with the *SuperTrans* simulator [155] that uses an abstract model of the TCC and Log-TM implementations for conflict detection and versioning. For all variants of HTMs exercising the proposed techniques, executing the STAMP and SPLASH benchmarks results in a reduced average energy delay squared product of $18\,\%$. Further experiments with synthetic workloads generated with *TransPlant* [154] reveal further potential with an average saving of $29\,\%$. Compared with work that proposed to serialize transactions [141], this approach reduces the average energy delay squared product by $30\,\%$. Moreshet et al. report that replacing locks with transactions for SPLASH2 [210] and running for 200 lock operations reveals that transactions need fewer cache and memory accesses and, thus, come with a reduced energy consumption [141, 140].

Ferri et al. propose an energy-efficient HTM on a cycle-accurate SW simulator [64, 63]. Their work evaluates the impact of cache structures and contention management schemes to find a trade-off between complexity, energy efficiency and performance in HTM systems.

From the presented works in this section, complementing the presented methods and strategies with a component that accounts for and weights specific optimizations against their impact on the energy consumption of the application seems an interesting research direction. Moreover, the models presented here could also help to refine the prediction of performance when applying optimizations.

3.7 Adaptive STMs

This section reviews works that achieve an adaptive STM system mostly through tuning STM internal data structures.

The dynamic performance tuning of data structures of the STM in TinySTM has been researched [62]. The dynamic optimization strategy tunes three important parameters: first, the hash function that maps a memory location to a lock is covered. Second, the lock array size is changed dynamically and third, the effects of resizing the lock array for hierarchical locking are studied. These dynamic tuning techniques target STM internal data structures and, hence, are orthogonal to the mechanisms for adaptation researched in this thesis. Thus, future work could even combine both techniques.

So far, the optimizations have global scope and, hence, affect all transactions of all threads. In [152], Payer and Gross research thread-local adaptive optimization strategies and propose *adaptSTM*. They artificially limit the optimization space through using eager lock acquisition. By this means, optimizations, such as changing the buffering of speculative values, become possible on a per thread granularity. Further, they research the impact of various hash functions and finally use a lock array of 2^{22} entries and 5 bits for the hash for the experiments. Their STM supports extensions of the read set size without aborting the running transaction through revalidation of the previously read addresses. The thread-local adaptive policies comprise write strategy, a configurable write hash array to

speedup look ups in the write set, a bloom filter for the same purpose, and a tuned hash function for the write array to increase data locality. These measures are implemented inside the adaptSTM and evaluated through running the STAMP benchmarks with TL2, TinySTM version 0.7.3 and 0.9.9. The comparison reveals that adaptSTM is faster than TL2 for all benchmarks except ssca2 and most benchmarks for both TinySTM versions. In the latter case the main performance gains are measured in case of an oversubscribed eight-core machine running with 16 threads. To conclude, adaptSTM performs as good as or better than TL2 and TinySTM in most cases. This approach relies on the sampling of additional counters that have been added to the implementation of the STM. The counters, that monitor transactions, aborts, commits, number of unique write locations, and read and write locations, describe the thread-local behavior of the last 64 transactions. The authors state that the overheads introduced through sampling and maintaining the counters are not significant.

Chakrabarti et al. introduce the Runtime Abort Graph (*RAG*) [32] that uses annotated memory references as nodes. An abort relationship is a directed edge from the node of the aborter to the node of the victim. The nodes additionally contain contextual information such as the atomic block, source code location, average read set and write set sizes, and the total number of aborts of this memory reference. The methodology works as follows: first, the compiler instruments the executable with counters for a training run in order to take samples. The program executes these points in case the counter reaches zero. The sampled information about the TM application is stored on disk. In a next post-processing step a tool reads this data and constructs the RAG. Based on a heuristic that takes the sizes of the read and the write set into account, the tool estimates the amount of wasted work locally for each contentious memory reference in an atomic block. In order to minimize the wasted work, a connection between the aborting and aborted transaction is established, which is called victim-aborter relationship. The information models the cost of the aborted victim locally and selects an acquire strategy for that atomic section (eager or lazy). A greedy algorithm combines these local solutions to find a global optimum for the acquire strategies. This is necessary because changing the acquire strategy can invert the victim-aborter relationship and the local model only optimizes for the victim which may find the best solution in a global scope. This information is fed back to the compiler which then changes the acquire policy. Hence, the execution of the newly compiled application now executes with a hybrid acquire policy. Different configurations for the training run (acquiring all references eagerly or all lazily) yield a different optimized setting after the profiling run. This shows that the optimizations are not converging yet to a globally best solution but improve upon the initial setting. For the other experiments the training and the reference data are identical. For selected STAMP benchmarks and across different processor counts, this approach yields an average improvement of $9\,\%$.

Bai et al. present a different way of adapting the utilization of an STM system [10]. Their work uses the executor classes of `Java` and DSTM [90] to build a key-based transactional memory executor. Instead of using the usual concept of threads that solve a specific problem and use transactions for synchronization, their scheme decouples transactions and so called work items through a producer/consumer pattern. The producer produces work items that the transactional memory executor assigns to a consumer and an available processor. Then the consumer executes the transaction on shared data. The assignment uses a specific key that contains information about the memory locations accessed in the transaction. For the experiments a custom function computes the keys for the transactions. This allows to schedule transactions that access the same memory locations on the same

	Top-down	Bottom-up
Online	Balasubramonian et al. [11] Huang et al. [96]	Balasubramonian et al. [11, 12] Duesterwald et al. [57]
Offline	Huang et al. [95, 96] Magklis et al. [130]	Shen et al. [181]

Table 3.1: Design space of phase detection, taken from [55]

processor executed by the same consumer. This not only limits contention but also exploits data locality through reuse of data present in the caches. Additional adaptivity of the TM executor is achieved through sampling the space of the keys and adjusting the range of the keys in case of an imbalanced distribution. On average the adaptive executor performs better than the fixed executor and both outperform a round-robin scheme. The price of this execution model are high overheads due to the producer/consumer pattern when compared with threads that execute transactions in a loop. This approach is interesting although it does not provide a universal solution for computing a key for an arbitrary transaction. Compared to the work presented in this thesis, this work makes fundamentally different assumptions about the execution of threads and transactions which make a fair comparison extremely difficult.

While these works describe important optimization techniques to fine-tune STM parameters at run time or using a profile and refinement step that involves the compiler, these techniques try to hide the actual run time behavior from the programmer. Although a programmer could simply use these approaches, these techniques could cover e.g., the bad layout of data structures and hide these from the programmer. Whereas tools that visualize the TM program behavior, as researched in this thesis, would help the programmer to eliminate the causes of degraded performance instead of covering them.

3.8 Phase Detection and Prediction

In preparation of the phase detection in TM applications, presented in Section 6.4, we give an overview over common techniques in the area of phase detection and prediction for sequential and parallel applications in the following. Note that these algorithms for phase detection and analysis are not related to TM.

Ding et al. [55] present a classic approach for phase detection and a classification scheme for related work as shown in Table 3.1. According to this scheme, Top-down and Bottom-up are distinguished as well as Online and Offline approaches. Top-down exploits knowledge of the source code to divide the program into *candidate* phases whereas Bottom-up applies metrics to identify recurring patterns in the execution. Online signifies that the analysis is performed at run time, whereas offline indicates that parts of the phase detection process are carried out after the program executed. Applying this classification to the proposed TM phase detection, we combine a bottom-up approach with offline phase detection, which is described in detail in Section 6.4. Sherwood et al. propose to take phase detection one step further by adding a run length encoding Markov predictor to predict program phases [184]. Discussed related work regarding phase detection covers uniprocessor execution. First work addressing phase detection in distributed shared memory systems has been conducted by Ipek et al. [104]. The work focuses on data distribution and contention in DSM systems.

In MPI applications, phase detection has been applied to obtain energy savings by frequency and voltage scaling [125]. Casas et al. study the application behavior with the goal to optimize the code and accelerate research in the scientific topic [26]. The digital signal is created by analyzing and aggregating time stamped events of trace files. The considered events are *Running* and *Idle* processor states. These are transformed into binary states (Running equals 1) and accumulated over all processors in the system. The *Wavelet* transform is then applied to the resulting digital signal.

Gamblin et al. employ a parallel wavelet transformation to identify load imbalance in very large parallel systems [70]. The wavelet coefficients are well suited for compression and, thus, combined with run-length and Huffman encoding. The parallel computation of the wavelet transform shows a near-perfect speedup [146].

Wavelet-based techniques are also applied to capture the memory bus behavior of commercial applications [98]. In particular a 2-D Haar wavelet is employed to characterize L2 misses. A calculation of a distributed wavelet transform is presented in [200]. Each transform stage is separated in three steps: split, predict, and update to enable a distributed calculation of wavelets in a sensor network. Wavelets are further part of the JPEG 2000 compression standard [187, 1].

Perelman et al. also present techniques for phase detection in parallel shared memory applications [153]. For each parallel thread a set of sampled BBVs (basic block vectors holding the execution count) is kept. Separate threads are regarded as independent entities. With transactional memory this approach is not sufficient anymore as the control flow directly depends on the interference with other threads. Due to the optimistic concurrency of the transactional memory model, transactions are executed speculatively and conflicts are detected. In case of a conflict one transaction is rolled back and executed again. As a consequence one thread directly influences the execution (in particular the control flow) of a neighboring thread. Hence, counting basic blocks executions does not sufficiently capture the phase behavior. In particular the reason for a repeated execution of a basic block is not revealed. Hence, TM-specific algorithms for the detection of phases in TM applications are required. We present two algorithms for the phase detection in TM applications in Section 6.4.

3.9 Open Questions with the State-of-the-Art

Section 3.1 reviews compiler support for TM and also presents approaches that combine compiler and run time system for optimizing the performance. There is a lack of open source compiler support for TM that is independent of the thread model in the GCC suite. Hence, we present design and implementation of an initial TM support in GCC in Section 7.1. In related work the run time system checks for specific properties to assure the correctness of compiler optimizations. The question that we long to answer is related: Can the compiler help to determine a specific property of the TM run time system? In Section 7.2 we present an approach that extracts and analyzes static information to select the conflict granularity of an STM system.

Section 3.2 summarizes approaches to extract information from TM and non-TM parallel programs. What are the pitfalls of generating TM event traces? May compression algorithms help to reduce the pressure on the write bandwidth of the hard disk? What are the advantages and drawbacks? We answer these questions in Section 5.1.

Section 3.3 shows related work that covers tools for TM and its optimization. None of these approaches includes the use of hardware performance counters into their work flow. How can we add the data from hardware performance counters into state-of-the-art visualization and optimization tools for TM? Can this information help the programmer of TM applications to gain additional insights? Chapter 6 covers these aspects.

Section 3.4 reviews profiling solutions with FPGA hardware. These do not cover the monitoring of different execution modes e.g., software and hardware in current hybrid TM systems yet. What are the challenges to track both modes of execution? Can special hardware alleviate the monitoring process and what are the associated costs in terms of hardware and probe effect for the TM application? How does this approach compare with the previous STM solution? Section 5.2 gives insights into the answers to these questions.

Moreover, current tools require that an inexperienced programmer repeats the steps for identifying conflicting data structures, modifying the program to reduce these conflicts and rerun the TM application to see whether the modifications yield benefits. How can we improve on this trial-and-error process that is the basis for the optimization of TM applications for the inexperienced programmer? Is there a way to extract the characteristics of the TM execution and use these to automate the optimization process? What are the challenges and drawbacks? Section 6.5 present first insights into these questions.

As described in Section 3.8, many algorithms for the detection of execution phases in sequential and parallel applications have been proposed and evaluated. These algorithms can not detect execution phases in TM applications yet. How can existing algorithms be transformed and modified to account for the specialties of transactional execution? What are the differences to local adaptation solutions which are presented in Section 3.7 and implemented in STM systems? Do the detected execution phases yield a potential for optimization through adjusting the TM algorithm? These open questions are answered in Section 6.4.

What are the advantages of applying TM to numerical algorithms e.g., the method of conjugate gradients? Can other algorithmic formulations help to bring out the strong side of TM? What can we learn from visualizing and comparing the two TM application's behaviors and their utilization of the microarchitecture? Section 6.3 illustrates our findings.

How is the performance of the first commercially available Hardware Transactional Memory system? What are the properties of a representative HPC benchmark to rate the synchronization with TM? What application properties take advantage of the optimistic concurrency with TM and how does TM perform with respect to other OpenMP-based synchronization primitives e.g., `omp critical` or `omp atomic`? Chapter 8 offers first performance numbers, a benchmark for TM execution in high performance computing, and a set of best practices for programming with a commercial hardware transactional memory.

What are the differences for tool support addressing HTM systems compared with STM or hybrid TM? How does a tool for measuring TM overheads of the BG/Q hardware TM system look like? How can performance counters on the BG/Q architecture be used to enrich the TM performance data and discover the cause for degraded performance? Chapter 9 displays how to achieve additional insights.

Section 3.5 highlights case studies that target the programming with TM. All of the students participating in the studies would have been possible candidates to try and evaluate the tools

for TM presented in this thesis. We believe that the achieved results would have been even better if tool support would have been available for them. Section 3.7 reviews adaptivitiy in STM systems. These have some tuning mechanism built in the TM system to optimize its performance during run time. Future work should also include the monitoring and visualization of these systems because an adaptive strategy may prevent the programmer from discovering principal weaknesses in the organization of the program's data structures. Future optimization objectives for TM applications could also include energy-awareness of the modifications and research how the results differ from using performance/execution time as objective function (cf. to Section 3.6).

4. Concept and Overview

This chapter contains a brief overview over the concept for optimizing TM applications used in this work. In contrast to the tool support for TM that has been described in related work that targets STM [7, 191, 127, 123, 220, 29, 76] or HTM [31, 71]. These tools only consider run time information specific for the TM system and source code information as input and are designed for a specific TM system that is tightly coupled with a programming language. We improve upon the state-of-the-art in two ways: first, we consider information from additional layers of the TM software stack and second, we direct our efforts not only at a single TM system but target TM systems implemented in software, hardware, or as a hybrid combination. Each of the TM systems comes with specific properties that are due to the type of implementation. These properties lead to different performance characteristics. The proposed methods and strategies for the optimization of TM applications must take these performance characteristics into account in order to support the application developer of TM applications.

Our approach extracts information from layers of the TM software stack that have not been considered until now to enable a broader scope for optimizations. In particular, we add static information and information retrieved from hardware performance counters to complement the information about the run time behavior of the TM application in order to obtain a better understanding of the utilization of the architectural resources. This information enables the programmer to learn more about the program behavior even in the absence of pathological TM execution patterns and can be useful in understanding the manifold side-effects of the TM system and the microarchitecture.

Hence, the approach of combining information from multiple layers of the TM stack, considering different TM systems and providing novel post-processing tools enables new insights into the behavior of applications and supports the programmer in optimizing these. Section 4.1 introduces the concept that comprises different levels of a TM software stack. Then, Section 4.2 illustrates the systems that we select for demonstrating and implementing the concept. Further, we highlight the relationship of the components to achieve the optimization of TM applications on the respective TM systems. Section 4.3 presents the experimental setups that serve as testbeds for the methods and strategies researched in this work.

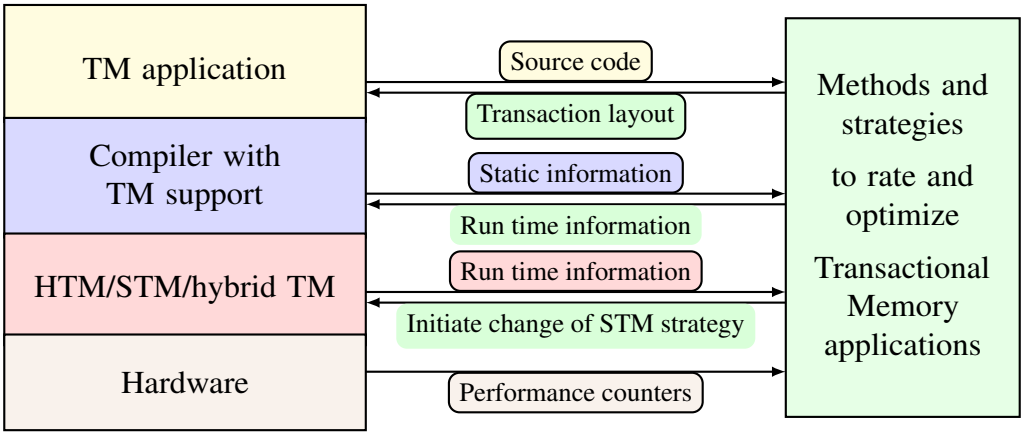


Figure 4.1: Schematic interaction of components in a system with TM software stack.

4.1 Concept for the Optimization of TM Applications

The left hand side of Figure 4.1 illustrates the necessary components for executing a TM application. The TM application defines the data structures and the parallel threads of execution. Parallel accesses to shared data structures require synchronization that may use Transactional Memory or pessimistic synchronization e.g., through the use of locks. In our work we focus on the use of Transactional Memory with optimistic concurrency. Optimistic concurrency does not enforce mutual exclusion of critical sections and, thus, requires a run time system to track accesses to shared data structures in order to identify and resolve conflicts. The TM-aware compiler transforms the memory accesses in transactions into calls into a run time system that detects conflicting accesses. Moreover, the compiler also generates the prologue of a transaction that checkpoints the architecture registers and starts the transaction. A TM run time system, implemented in hardware, software or as hybrid combination of both, tracks these memory accesses and resolves conflicts through aborting a transaction. These TM run time systems may either be executed on top of commodity hardware or require minor to profound changes of the processor architecture.

The different layers of the TM software stack provide different information that we propose to combine in methods and strategies to rate and optimize Transactional Memory applications (see right hand side of Figure 4.1). The application level provides information about the source code e.g., code line and context of the execution, the TM compiler supplies static information e.g., instrumented memory locations, the TM run time system adds information about the TM behavior e.g., TM statistics or event logs, and the hardware offers hardware performance counters for performance monitoring. Combining the information from the different layers in methods and strategies enables a better understanding of the TM behavior as well as optimizations that go beyond those triggered only through monitoring a single layer in isolation. In order to characterize the performance of the TM system and give advise to the programmer of TM applications, we need a profound benchmark that mimics real applications and uses TM as well as other synchronization primitives to compare their execution times. Especially for high performance computing, we need to establish a set of best practices for programming with TM. From processing the information, the strategies may indicate certain actions that affect the performance. On the source code level, the methods and strategies may point to a change of the transaction layout to benefit performance. Static information, gathered through the compiler and analyzed in specific passes, may guide the selection of an STM parameter to obtain a better performance

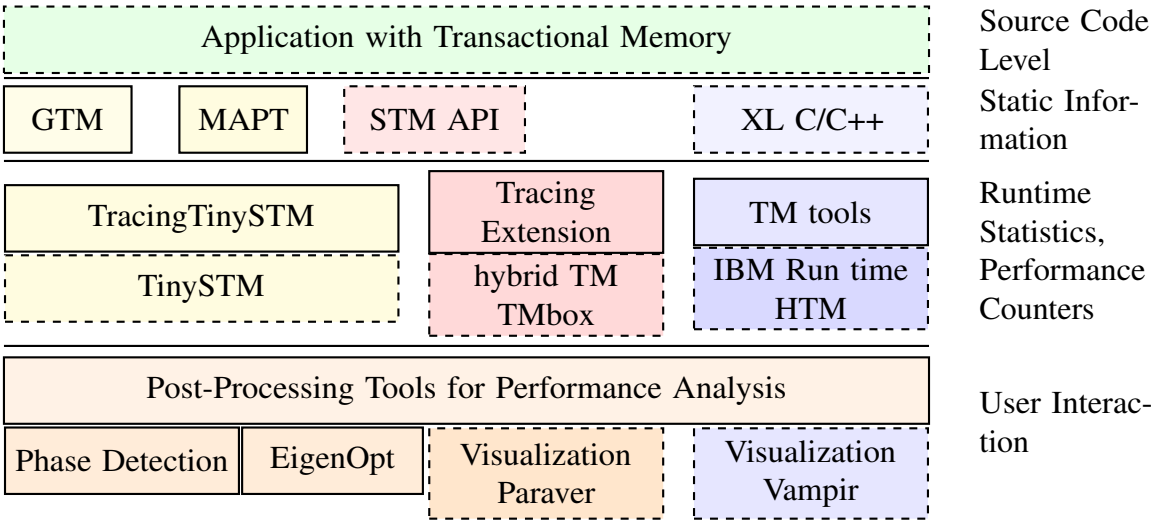


Figure 4.2: Overview and relationship of components presented in this thesis.

of the application. On the level of the run time system, e.g., selecting a different STM strategy for conflict detection may be indicated through the methods and strategies. The foundation for these methods and strategies provides the retrieving of detailed run time information of the TM system. The information enables the programmer to understand when transactions started, committed, or aborted. Complementary information from the hardware performance counters enables to monitor the utilization of the microarchitecture and e.g., discover stall cycles due to cache misses and relate these to the execution of transactions.

4.2 Components that Implement the Concept

In the following we describe the components that implement the concept presented in the previous sections and give an overview over the TM systems that we selected to demonstrate the tools for TM. While the concept is general enough to be implemented for various TM systems, we select an STM, a hybrid TM, and a HTM system and exemplarily design and implement tools for each of these systems. Figure 4.2 illustrates the relationships of the components that implement the concept for the optimization of TM applications. In the figure all components that we contribute to the state-of-the-art are framed with a black solid box whereas the work of others exhibits a dotted frame.

In our concept, the TM application rests on one of three pillars: STM, hybrid TM and HTM. These three systems enable to compare the different possibilities for optimization that are enabled through the run time information that we retrieve.

The source code has to address each of the run time systems in a slightly different way. Applications that target STM either use a pragma/keyword or are hand-instrumented with the STM API. An extension of the `C` language with a pragma is achieved with the GNU TM (GTM) approach that presents the initial basic support for STM in the GCC. With this work, we contribute to the open source community efforts around GCC and enable researchers to experiment with STM. From this experience with instrumenting memory accesses, we derive the Memory Access Patterns in Transactions (MAPT) approach that uses static information to propose an STM property.

Both compiler extensions address TinySTM, a word-based STM that is available as open source. We augment this STM with a machinery for event logging. During run time, TinySTM generates event traces that capture the application's behavior at a fine granularity. Hence, we named the extension TracingTinySTM. TracingTinySTM accesses hardware performance counters through the PAPI interface to enrich the TM-specific events with architecture-specific events. The post-processing tools either transform the traces into a different format for a visualization tool e.g., Paraver or perform an advanced analysis such as the detection of phases in the TM application.

As a hybrid TM system, we use the FPGA-based TMbox system. We extend the hardware of TMbox with an event generation and an event logging unit that enables to trace events at a granularity that is even finer than that of TracingTinySTM. The event traces contain information about the execution in hardware and software. A post-processing tool generates aggregate statistics and converts traces for the visualization with Paraver.

The HTM system is IBM's proprietary BG/Q architecture. The proprietary IBM XL C/C++ compiler provides the compiler and run time system for this architecture. The compiler transforms a special pragma, that marks a transaction in the code, into calls to start and commit a transaction. The run time provides information about the TM behavior as summary statistics. With these preconditions, we design a tracing and a profiling tool that correlates these statistics with the BG/Q hardware performance counters. Vampir, a state-of-the-art visualization tool, visualizes the snapshots obtained with the tracing tool.

In order to establish a set of best practices for programming with TM on BG/Q, we need a benchmark that is simple to use and has sufficient parameters to generate different application's behaviors. For high performance computing, we present CLOMP-TM, a benchmark that compares synchronization with TM with other OpenMP-based synchronization primitives in order to select the fastest. Due to its versatility, CLOMP-TM helps us to correlate a certain TM behavior with the corresponding performance and derive a set of best practices for programming with TM.

The choice of the TM run time systems enables us to compare the TM architectures from different ends of the spectrum. Comparing an open source STM, a highly customizable FPGA-based hybrid TM system and a proprietary HTM system yields insights how much information must be available for a post-processing tool and what kind of optimization yields a better performing TM application on each of the systems.

4.3 Experimental Setup

In the following we highlight our primary experimental platform for STM experiments. The `x86_64` architecture used in our experiments comprises two Intel® Xeon CPUs (called Westmere) each with 6 cores and hyper-threading technology. Westmere implements the Nehalem microarchitecture and is implemented in 32 nm technology. It features 24 hardware threads in total. Each core has an exclusive L1 and L2 cache whereas L3 caches are shared inside a socket. Table 4.1 presents the sizes of the caches and the CPU frequency. Hereafter we will refer to this architecture as *ExpX5670*.

A second setup is an Intel® Core2 Quad Processor Q6600 with 4 GBytes of main memory, 8 MBytes L2 Cache, 2.40 GHz and no hyper-threading hereafter denoted with *ExpQ6600*.

A third setup is an eight core Intel® Xeon® E5410 running at 2.33 GHz, with a L2 cache size of 6 MBytes with 4 GBytes of main memory and no hyper-threading. We refer to this system as *ExpE5410*.

Intel® Xeon® CPU	X5670
CPU frequency	2.93 GHz
Processing Units per Core	2
L1 size	32 KBytes each
L2 size	256 KBytes each
L3 size	12 MBytes each
Number of Cores	12 total
Number of Hardware Threads	24 total
Number of Sockets	2

Table 4.1: Experimental platform ExpX5670.

A fourth setup constitutes of a two socket quad-core with a total of eight AMD Opteron™ cores each being of type processor 2378 clocked with 2.4 GHz, private L1 caches with 64 KBytes, private L2 caches with 512 KBytes, and L3 caches shared inside a socket with 6 MBytes each and 16 GBytes of main memory. This system is named *Exp2378*.

5. TM-specific Trace Generation for STM and Hybrid TM Systems

This chapter introduces TM-specific solutions to log events in STM and hybrid TM systems. A carefully designed logging mechanism is required to record the genuine TM application's behavior and preserve it for subsequent post-processing steps. The first solution targets STM systems and describes the difficulties and challenges that logging events introduces in the TM setting and is similar to [175]. Section 5.1 presents the details of the implementation that employs compression algorithms and uses multiple threads for compression. The second solution aims at hybrid TM systems and is tailored to the FPGA-based TMbox system. Section 5.2 describes the design and implementation of adding logging mechanisms to the TMbox system architecture, as described in [189, 112]. Section 5.3 compares the run time and the area overhead of both approaches. Section 5.4 concludes this chapter by comparing both event logging architectures and summarizing the findings.

5.1 Augmenting TinySTM with Trace Generation Facilities

Event traces record the temporal occurrence of events initiated through the application. Information retrieved from these event traces is often used to improve the application's runtime behavior (e.g., find memory access patterns with higher spatial and temporal locality). One key point is that obtaining these event traces does not change the application behavior, which is called non-intrusive tracing. In some experimental setups, the goal of non-intrusiveness can be achieved with smaller costs than in others. For instance, the coarse-grained nature of MPI tasks makes them generally amenable for profiling with library calls intercepting the original MPI calls (e.g. PMPI). Since MPI events occur rather infrequently and mostly at the beginning or end of computation-intensive regions, the trace generation does not significantly alter the program behavior. In the context of parallel threads, generating event traces is not as straightforward as in the MPI case. Unfortunately, a perfect solution for generating traces does not exist. We research and evaluate how to ameliorate the costs of logging events in a TM context. These efforts are motivated by the

discussion of related work, presented in Section 3.2. Current tracing solutions are either not specifically tailored for TM and may have higher overheads than required for tracing TM applications (e.g., [113]) or take approaches that introduce locks on the critical tracing path and can not leverage the full potential of compiler optimizations [29].

5.1.1 Minimizing Application Disturbances

The trace generation introduces overhead when logging events during the runtime of the application. This overhead may influence two important performance metrics of a TM application: throughput and number of aborted transactions. The non-uniform delay of threads when logging events may influence both metrics. Delayed threads may lead to additionally detected conflicts. With TM this effect is more severe than with mutual exclusive synchronization: A conflict leads to a rollback of one transaction and all previous modifications of the transactions are undone and recomputed. In case this conflict is artificially generated through delaying the thread by the tracing machinery, the recorded application's behavior is not genuine. Hence, avoiding additional delays is one goal. Threads may be delayed when writing the logged events from a buffer to the hard disk. In prior work Knüpfer et al. propose to store a trace of n threads [processes] with $1...n$ streams [113]. If the number of streams is less than n, additional synchronization is introduced to coordinate two threads that write to the same stream. In our use case this synchronization comes with the undesired side-effect of artificially delaying threads. Therefore, we only allow a bijective mapping of n threads to n streams – e.g., each thread writes to a separate file. In contrast to previous approaches [113], the trace generation is not encapsulated inside a dedicated library due to performance reasons. The STM library already stores meta data in thread local storage. Through extending the meta data and manually inserting calls to log events, we achieve a higher data locality. Further, the compiler may inline and optimize the tracing code and achieve a higher performance compared with call back-based approaches. The interface of the TinySTM remains unchanged, thus, this tracing approach is also transparent to the application developer. The event logging mechanisms are integrated in the build process of the STM in a way that these are easily accessible for users.

The design choices follow the need to preserve the original course of events. In order to reduce the interference between threads, each thread writes to a separate trace file. The corresponding file handles are stored in the thread local storage. Also, we add functions to read the OS time, cycle counter (TSC) and PAPI events. PAPI is a platform independent interface to access and control hardware performance counters [139]. In our setting PAPI enables us to relate the utilization of the microarchitecture, monitored with the performance counters, to specific transactions.

Event traces may be captured in two different formats: as ASCII or as binary packets. The format of the binary packets is shown in Table 5.1. The transactional events are captured by instrumenting the respective STM functions. We took care when placing the instrumentation such that events will not be logged prematurely: e.g., after entering the function `stm_commit`, the transaction may still abort during the validation of the read and write sets. The PAPI events are optional and read the hardware performance counters defined during compile time. These measurements correspond to the execution of a transaction (from start to abort or commit). The ASCII mode introduces one `fprintf` call per event. This first approach writes elements of varying size at once whereas the binary tracing approach buffers events of fixed size in a dedicated local memory buffer.

Type 8 Bit	Payload 64 Bit	Payload2 64 Bit
Timing1	tx [%p]	OS time [sec]
Timing2	OS time [nsec]	padding
RTSC	TSC [cycles]	padding
Start	tx [%p]	STM counter
Read[1]	address	padding
Write[1]	address	value
Commit	tx [%p]	STM counter
Abort	tx [%p]	STM counter
PAPI1	counter value	padding
PAPI2	counter value	padding

Table 5.1: Format of timing and transactional events in x86_64 binary trace files with optional reading of performance counters through PAPI. Extended version of table in [175].

The buffer size is given in number of event elements. A $10K$ element buffer signifies that each thread allocates a buffer of $136[bit] * 10K \approx 166KBytes$.

5.1.2 Implication of Lightweight Trace Generation on Offline Analysis

The goal of a lightweight tracing scheme is to reduce the runtime disturbances of the application. A trade-off between logging heavy-weight events with high precision and lightweight events with low precision must be found. Then, an advanced offline analysis can reconstruct the original course of events. Having a precise notion of time is mandatory for visualization and especially for finding causalities among the traced events. In the following, we propose to reconstruct the time line of events based on the very scarce usage of a heavy-weight time stamp and the frequent logging of a lightweight cycle counter. In contrast to vector clocks [65], that capture the causality of events and preserve a partial ordering, our approach interpolates the real time when an event occurred. This enables to order events even when both events do not have a causal relationship. The function `clock_gettime` delivers the time synchronized across all processing cores (heavy-weight). At Start, Abort, and Commit events, the cycle counter of the core is logged (with the `RDTSC`[1] instruction). Despite being lightweight, this counter comes with the disadvantage of not being synchronized across cores. Through also logging it at thread init and exit, both time stamps are correlated. During offline processing, the time between two events (e.g. thread init and start) is calculated according to $\Delta t = \frac{\Delta c}{f}$ where f is the frequency of the core and Δc is the number of cycles between the two events. By adding Δt to the heavy-weight synchronized time stamp (at initialization of the thread), the real time of each event is reconstructed. Transactional events in between start and commit are interpolated by assuming that each event consumes an equal amount of time. Further, repeating these steps for all threads, enables to establish an order of the concurrently generated events.

5.1.3 The Influence of Tracing on the Runtime

In the following, we will use a TM application, called `bank`, to study the influence of the tracing machinery on the runtime of the application. `bank` manages a fixed number of bank

[1]RDTSC is not influenced by dynamic voltage and frequency scaling of the cores.

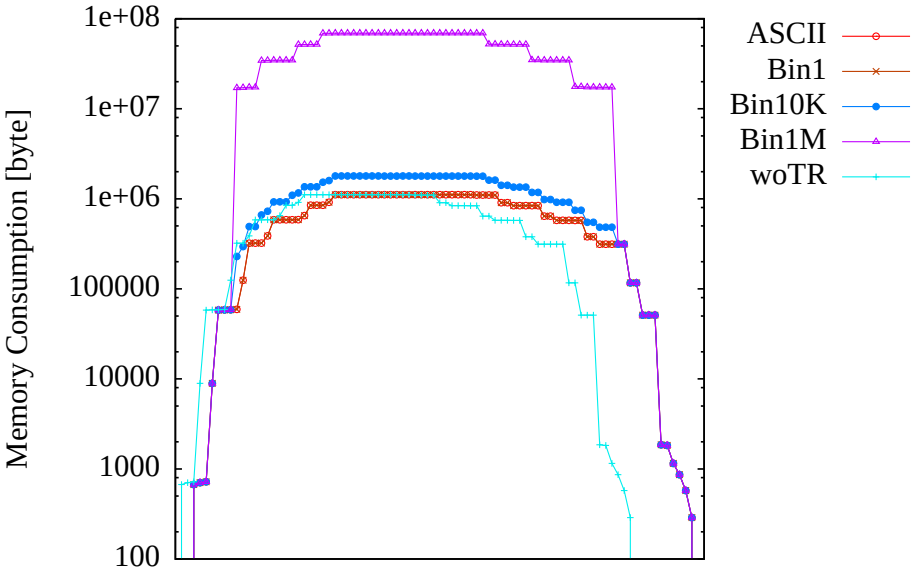


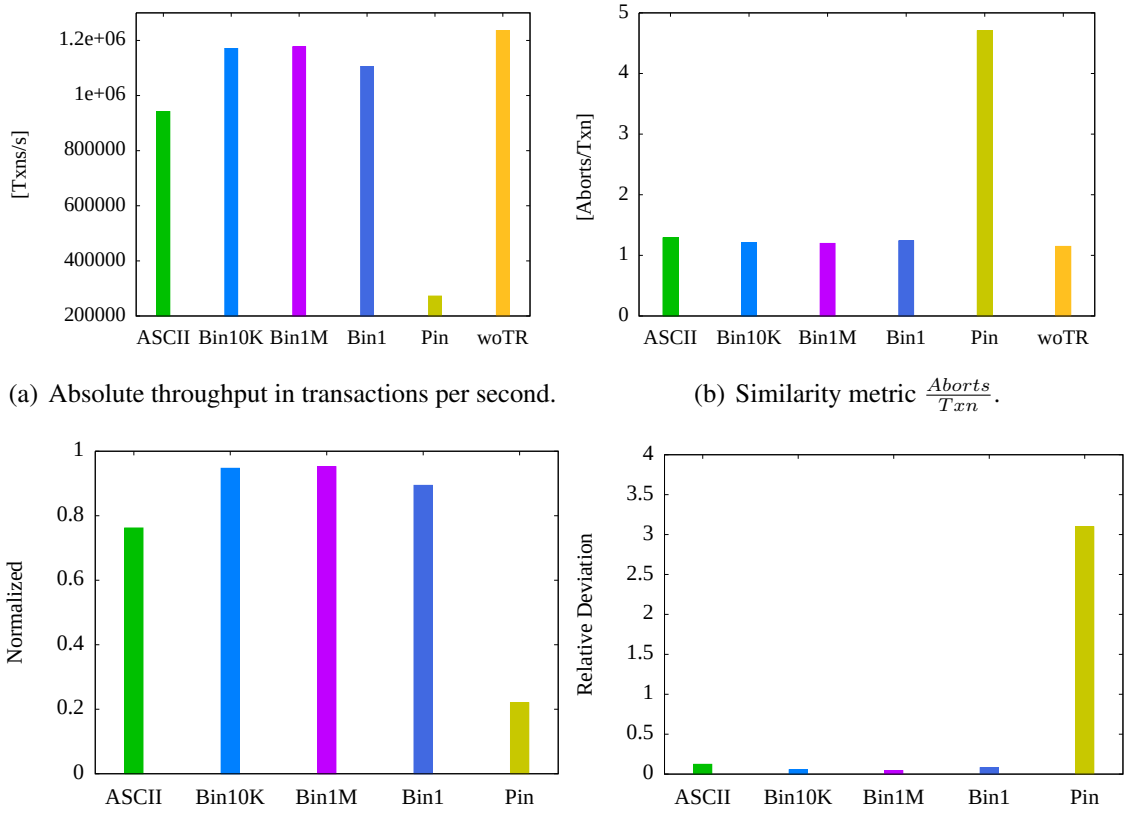
Figure 5.1: Heap size of `bank` application (from [175]).

accounts and transfers money between them with transactions. Long running read-only transactions are carried out in turn with short writing transactions. Since `bank` spends most time inside transactions, it is a useful stress test for the tracing machinery because of the high write bandwidth demands. Hence, `bank` is chosen to study the influence of the trace generation on the memory consumption and TM metrics.

Memory consumption at runtime is illustrated in Figure 5.1. More precisely the graphs represent the heap size of the running application with different tracing variants. These results are generated with `valgrind` [145] in particular employing the tool `massif` to sample the memory consumption. The x-axis shows the sampling points in time where the heap size is determined. The memory consumption without tracing (woTR), ASCII tracing (ASCII), and binary tracing with different buffer sizes: 1 element (Bin1), $10K$ elements (Bin10K) and $1M$ elements (Bin1M) are shown on the y-axis in a logarithmic scale. Bin1M dominates all variants with 70 MBytes memory consumption at most. The average heap consumption for the pure application `bank` is less than 1.2 MBytes. Increasing the buffer size to $10K$ elements increases memory requirements to 1.8 MBytes. The memory demands are negligible compared to the available memory in most desktop machines (around 4 GBytes). The increased memory requirements lead to a larger cache footprint which may have a severe impact on the application as it may lead to more cache misses.

In order to demonstrate the adequateness of the increased memory requirements, we compare our customized trace generation with a state-of-the-art dynamic instrumentation tool called Pin [129]. Instrumentation is achieved through so called Pintools which are written in `C/C++`. At runtime Pin uses binary instrumentation to connect the unmodified application and the Pintool. A Pintool usually consists of two parts: an instrumentation and an analysis part. In our case, the instrumentation registers call backs for the STM functions[2] start, commit, abort and thread create and exit. The analysis part logs the events

[2]For full traces transactional loads and stores are also instrumented.



(a) Absolute throughput in transactions per second.

(b) Similarity metric $\frac{Aborts}{Txn}$.

(c) Throughput normalized to execution without tracing.

(d) Relative deviation of the metric $\frac{Aborts}{Txn}$ from execution w/o tracing.

Figure 5.2: Influence of trace generation on the TM behavior. Figure taken from [175].

and writes them to a file. However, not all events are available with the Pintool - only the ones underlined in Table 5.1 are logged. At thread create and exit the trace files are opened and closed respectively. In order to enable a fair comparison to our approach each thread writes to a separate file directly and, hence is comparable with ASCII or Bin1.

Influence on the application's behavior as mentioned before, shared memory applications, featuring separate threads of execution are sensitive to modifications. Because these modifications are necessary to generate event traces, the overhead of generating event traces and the influence of the experimental setup on the application's runtime behavior must be studied. As a first indicator which combination of tracing and experimental setup is suited, we investigate the influence on the throughput of the application. The application `bank` runs for a fixed period of time (by default 10 seconds). During this time, the amount of transactions executed per second is measured. Figure 5.2(a) depicts this throughput. Correspondingly, Figure 5.2(c) holds the normalized throughput, which is computed according to $\frac{Txns_X}{Txns_{woTR},}$ where X selects the tracing variant and $woTR$ means execution without tracing. These performance numbers show that the Pintool limits the throughput in transactions per second to $22\,\%$ whereas all proposed variants reach more than $76\,\%$.

Further, we seek to quantify the influence of tracing on the TM characteristics. Due to the optimistic nature of transactional execution, transactions may conflict with each other. Resolving these conflicts usually leads to an abort of one of the transactions. When combining the rate of aborts with the rate of transactions, a metric is found that serves

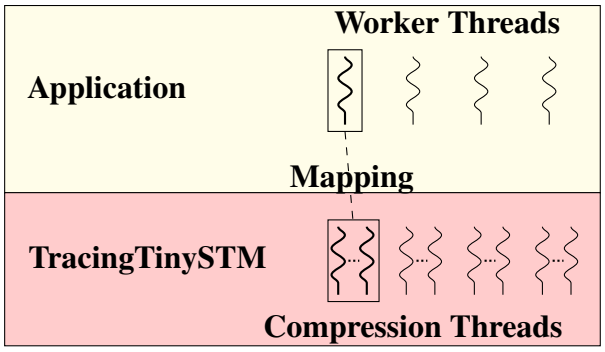


Figure 5.3: TracingTinySTM with support for online trace compression [175].

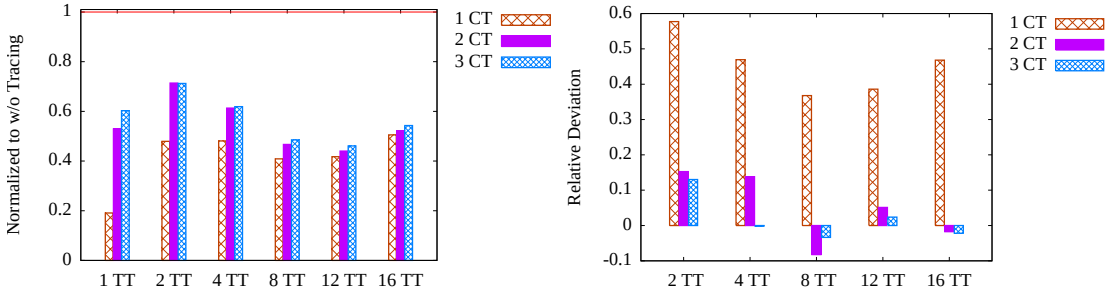
as a measure for similarity of transactional execution (computed as $\frac{Aborts}{Txn}$). In case the application is not affected by the tracing machinery, the metric yields similar values because the amount of aborts experienced per transaction is similar. Figure 5.2(b) shows this similarity metric for our tracing variants and Pin. Computed according to the formula $(X - woTR)/woTR$, Figure 5.2(d) demonstrates the relative deviation of the tracing variants compared without tracing. Since a smaller deviation means similar program behavior, our tracing variants preserve the application's behavior better than Pin.

5.1.4 Online Trace Compression

In a multi-core system with a large number of threads executing and generating traces concurrently, the write bandwidth of the hard disk soon becomes the bottleneck. Therefore, compressing the trace data prior to writing it to disk seems a viable solution. However, compression algorithms have to be carried out by or on behalf of the application thread. This poses a major challenge if the application is not to be influenced by the compression. As one of our declared goals is to minimize the disturbance of the application, having the application thread compress the data is counterproductive. While carrying out the compression the thread would execute non-transactional code instead of playing its original role with the other threads. Therefore our approach decouples trace generation (done by the application or worker threads) and trace compression (carried out by compression threads). This basic principle is also illustrated in Figure 5.3. Due to the API of the TinySTM, compression threads are spawned and destroyed transparently to the user. E.g., when a thread initializes its transactional state, a predefined number of compression threads is created. These threads will then compress a buffer of fixed size when signaled by the application thread. However, writing or compressing a buffer takes different amounts of time. Thus, a suitable mapping of application to compression threads must be found.

In [113] the use of the ZLIB library[3] for compression has been proposed. Our first experiments revealed that a thread mapping of approximately 1:10 would be necessary to keep the application thread from waiting. Thus, our approach uses the slightly different Lempel-Ziv-Oberhumer (LZO) library which is designed for real-time compression[4]. Although the additional speed comes at the cost of a lower compression rate, the results are still satisfying. The LZO library compresses $4,607$ MBytes to 156 MBytes yielding a compression factor of 29. However, this is also due to the fact that the binary format contains padding which compresses easily.

[3] J. Gailly and M. Adler, zlib, `http://www.zlib.net/`
[4] M. F. Xaver and J. Oberhumer, LZO, `http://www.oberhumer.com/opensource/lzo/`



(a) Normalized to execution without tracing.

(b) Relative deviation of Aborts per Transaction compared without tracing.

Figure 5.4: Throughput and similarity with multi-threaded trace compression (as in [175]).

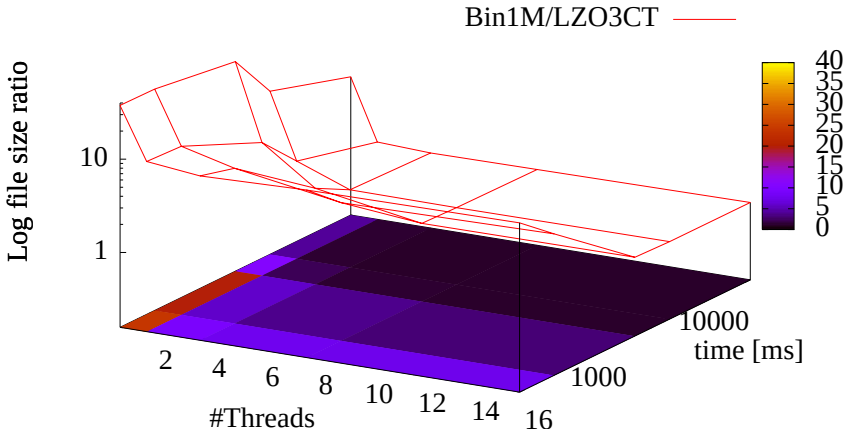
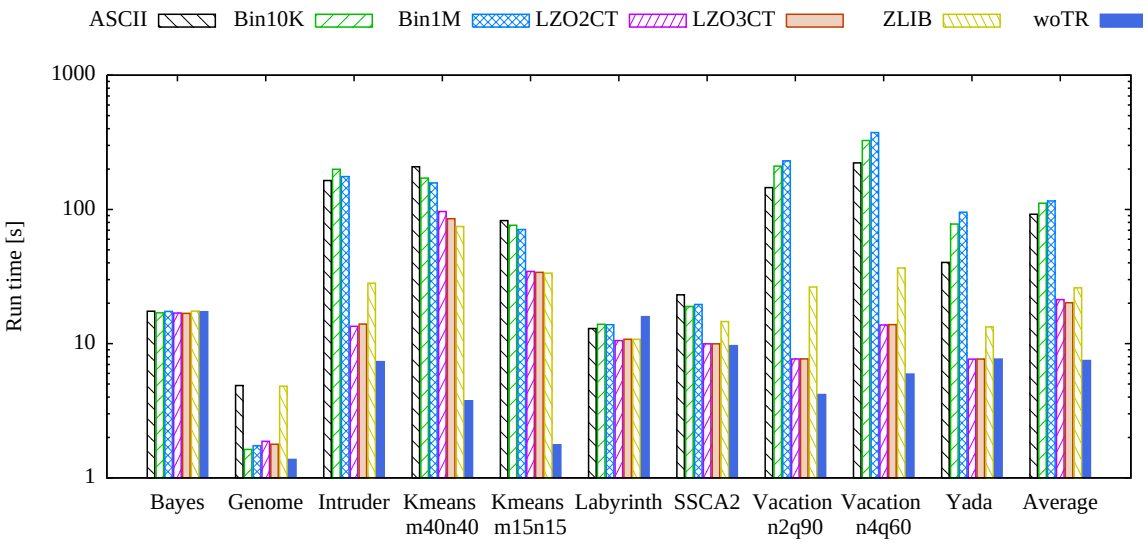


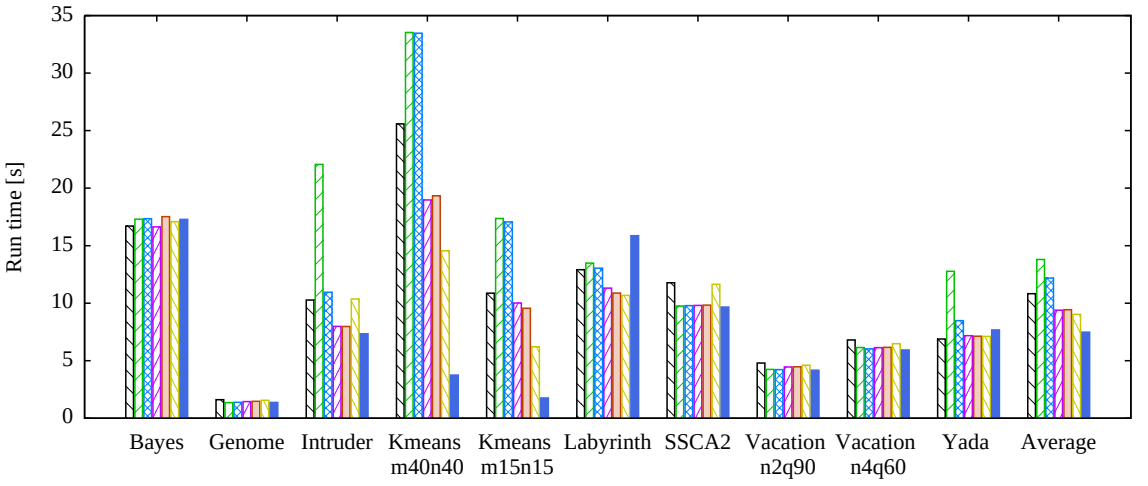
Figure 5.5: Compression factor computed as $\frac{Size_{Bin1M}}{Size_{LZO3CT}}$ as a function of thread count and computation time (with logarithmic z scale).

The throughput of the application threads executing without tracing is compared with different mappings of application to compression threads for trace compression. E.g., 2CT signifies that 1 application thread maps to 2 dedicated compression threads. These compression threads compress data inside a buffer by calling the LZO library. 1CT and 3CT are constructed accordingly. Figure 5.4(a) shows the Txn/s normalized to execution without tracing. This throughput ranges between $44\,\%$ and 71%. This is a serious drop when compared to tracing without compression which can be ascribed to the larger memory footprint. More buffers need to be utilized by the tracing thread and executing the LZO routines also has an impact on the instruction cache. Further, more threads share the same computational resources which are eventually saturated. We argue that the similarity in the application behavior is more important than the measured throughput. Figure 5.4(b) shows that all tracing variants except 1CT capture the application behavior in an acceptable way because the relative deviations of the $\frac{Aborts}{Txn}$ are smaller than $15\,\%$. A negative deviation means that execution without tracing has a lower $\frac{Aborts}{Txn}$ rate. Therefore, the setups with LZO2CT and LZO3CT are adequate for trace generation.

The compression factor of the LZO3CT scheme compared with the Bin1M setup is shown in Figure 5.5. The logarithmic z-axis holds the compression factor, computed as $\frac{Size_{Bin1M}}{Size_{LZO3CT}}$. The y-axis displays the number of threads and the x-axis the execution time of



(a) Tracing all TM events (includes transactional loads and stores).



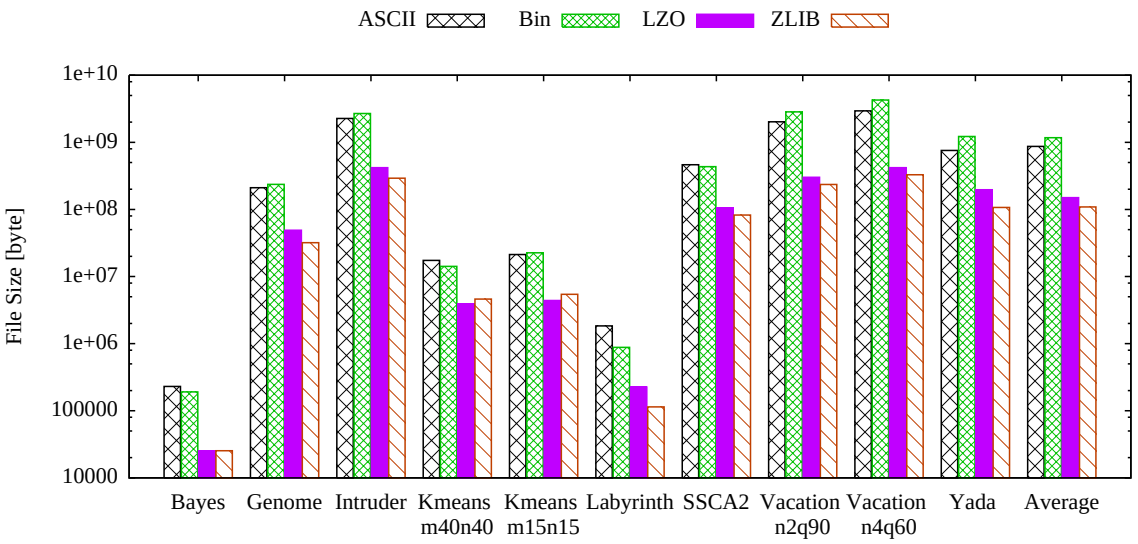
(b) Tracing a reduced set of TM events.

Figure 5.6: Average execution times of the STAMP benchmarks. Previously published in [175].

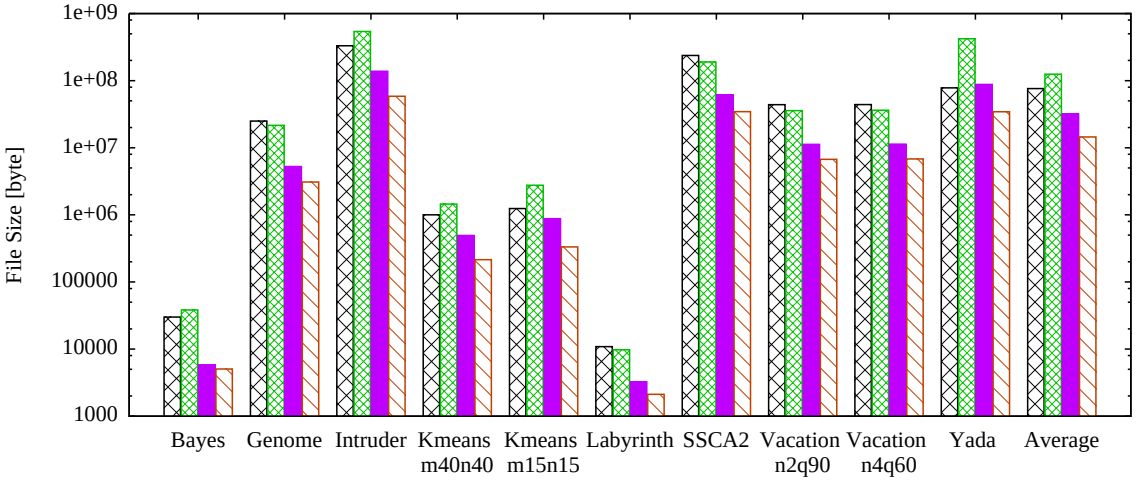
the benchmark. LZO3CT efficiently reduces the size of the log files when the thread count is low. Further, the runtime of the `bank` application should not exceed 5 s to benefit from reduced file sizes. This time span is comparable with simulators for TM that run for one day.

5.1.5 Impact of Trace Generation on STAMP Benchmarks

In this section, we study the impact of the proposed trace generation schemes on the execution of TM benchmarks that stem from multiple application domains. For our experiments all benchmarks from the STAMP TM suite [24] run consecutively and to completion on the experimental setup ExpX5670 (cf. to Section 4.3). The tracing functionality is implemented inside the TinySTM (version 0.9.9) [62]. Each of the 10 STAMP benchmarks solves a problem of fixed size. All reported values are averages over 30 runs to compensate for variations in the execution. Each benchmark runs with 8 worker threads. The thread count is sufficient because the STAMP applications show a limited scalability on this architecture.



(a) Tracing all TM events with TM loads and stores.



(b) Tracing a reduced set of TM events.

Figure 5.7: File sizes for traces of the STAMP benchmarks [175].

The trace scenarios are as follows:

- *ASCII*: traces are in the ASCII format without buffering,
- *Bin**: binary traces with a 10K and 1M element buffer,
- *LZOkCT*: LZO compression with k compression threads for each worker thread,
- *ZLIB*: ZLIB compression without compression threads,
- *woTR*: without any tracing enabled.

Figures 5.6(a) and 5.6(b) depict the execution times of each individual STAMP benchmark with and without tracing and the computed average over all benchmarks. The average shows the largest increase in runtime for the ASCII and plain binary tracing variants. The `labyrinth` application surprisingly shows a speed up when tracing is enabled. The TM statistics between traced and w/o tracing do not differ significantly. As a consequence, a biased TM behavior does not cause this speedup. The exact reason is still unclear but further investigations reveal that the runtime of `labyrinth` has a high variance and the

Setup	L3 cache misses		
	Min	Max	Average
woTR	$1.7\,10^7$	$2.1\,10^8$	$9.9\,10^7$
Bin1M	$1.1\,10^7$	$1.6\,10^8$	$4.8\,10^7$
LZO2CT	$1.2\,10^7$	$1.2\,10^8$	$2.8\,10^7$

Table 5.2: L3 cache misses for `labyrinth` (cf. to [175]).

number of L3 misses is lower for the tracing variants. The Table 5.2 holds the range and average value of these misses.

However, `labyrinth` is an unusual case that eventually leads to a small speedup. The common case is e.g., `kmeans` that shows that the tracing threads are competing with the application threads for architectural resources. This competition leads to additional stall cycles and increases the cache miss rate; eventually contributing to an increased execution time. Execution times increase substantially, when *all* transactional events are traced (cf. to Figure 5.6(a)). Please note the logarithmic y scale, which also emphasizes the prolonged times of ASCII and Bin*. Herewith, the benefits of the online compression approaches are demonstrated. On average LZO3CT yields a $30\,\%$ better runtime than ZLIB and $451\,\%$ better than Bin10K. The runtime effects are not as prominent in Figure 5.6(b), when tracing only a *subset* of the TM events.

Figure 5.7(a) and Figure 5.7(b) show the average sizes of the trace files with and without compression. Again, please note the logarithmic y-axis. Regardless of the amount of data, the ZLIB library provides a higher compression ratio on average but also records a potentially biased TM behavior because application-level threads also perform the compression. For the small event set, the LZO library yields a compression factor of 3.94 compared with 8.63 for ZLIB. For the large event set, LZO compresses with a factor of 7.82 and ZLIB with a factor equal to 10.79 on average.

5.2 Event Logging in a Hybrid TM System (TMbox)

In the following, we will demonstrate how a hybrid TM system can be extended with event logging facilities and compare this approach with the STM-centric trace generation. *TMbox* is a hybrid TM system that builds on Field Programmable Gate Arrays (FPGAs) [190]. In an FPGA, reconfigurable logic connects lookup tables, flip flops, block rams, and hard cores (e.g., ethernet cores or units for digital signal processing). Programming the FPGA connects and configures these logic components in a way that is described in the design process. TMbox exploits this flexibility and configurability to provide an excellent testbed for Transactional Memory research. The reconfigurable logic enables to experiment with different buffer sizes and explore design trade-offs. For designing and developing a low-overhead extension to log TM events, the setup of the TMbox system is well suited.

Figure 5.8 illustrates the hardware part of the TM system. A ring bus connects 8 Honeycomb soft cores and the bus controller. In this original configuration 16 cores fit on one Virtex-5 FPGA XC5VLX155T. The cores support the MIPS R3000 instruction set architecture and have been extended with special TM instructions: `XBEGIN`, `XCOMMIT`, `XABORT`, `XLB`, `XLH`, `XLW`, `XSB`, `XSH`, `XSW` and `MFTM`. These instructions start, commit, abort, or respectively load and store memory values in transactions or move data from registers. These instructions trigger state changes in a finite state machine that monitors

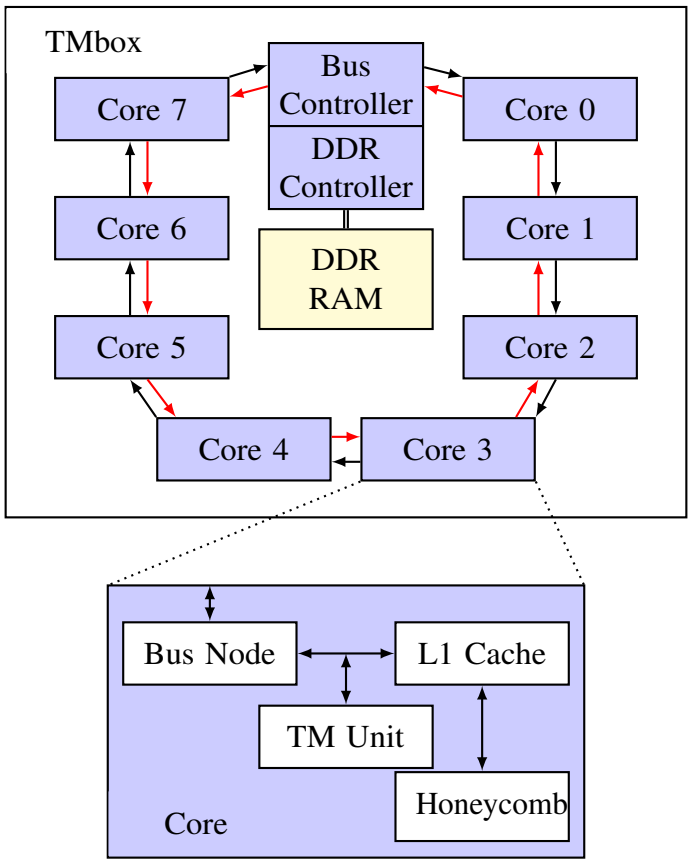


Figure 5.8: Overview of the TMbox system, showing a bi-directonial ring bus that connects 8 cores and a bus controller (similar to [190]).

whether a transaction is ongoing or not. On a higher level these introduced states appear as *TM Unit* which is also illustrated in Figure 5.8. The TM Unit accommodates a cache with up to 16 entries for transactional loads and stores (these are not held in the L1 cache). Further, the TM Unit snoops the bus for invalidations. Upon receiving an invalidation to an address in the read or write set, the running transaction is aborted and possibly retried. This elegant solution uses two ring buses: one for requests and responses to/from memory and one for invalidations from writes to memory. Because both buses operate in opposite directions (also confer to Figure 5.8), an invalidation may cancel a write request on any node if both carry the same address. Three new bus states enable to execute a commit operation with multiple writes: TMbusCheck, TMlockBus and TMwrite. These assure that a commit operation is not interrupted through other bus requests because it locks the bus first. In HTM mode a transaction may not occupy more than the 16 entries of the TM cache. Thus, an STM system is added to the hardware to relax this restriction. The TinySTM-ASF, a hybrid version of TinySTM designed to work with AMD's ASF extension, is ported to TMbox. Using the hybrid TM system in the hardware mode effectively reduces the number of cache lines available for data because ASF requires to also hold the lock address in the TM cache. If both are stored in the same cache line, the HTM performance is not affected. The experimental results verify that pure HTM performs faster than Hybrid TM or STM as long as the transactions fit in the TM cache. The last paragraph contains only a short introduction to TMbox. For more information please refer to [190]. In the following, we will highlight how we enhance this TMbox architecture with low-overhead mechanisms to generate event logs.

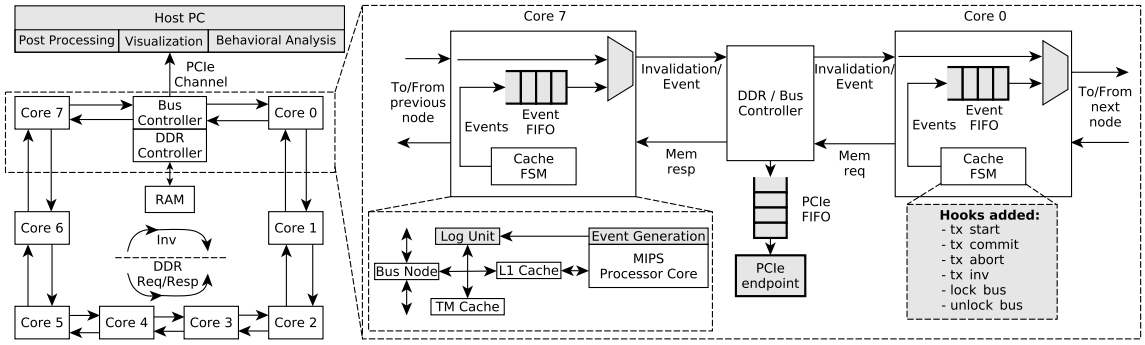


Figure 5.9: An 8-core TMbox system block diagram and modifications made to enable logging of events (shown in gray) – taken from [189].

Similar to the previously introduced tracing methodology for STMs, this approach aims at imposing low-intrusiveness on the TM application through having extremely low run time overhead [189, 111, 112]. Additionally the design focuses on low-area overhead, meaning that only a minimal amount of FPGA logic should be added for logging events and that the added logic should not affect the place and route process of the existing components in a way that these do not reach their original frequency. This could happen if the added components negatively influence the critical path of the design and, thus, lower the achievable frequency. A special requirement for monitoring a hybrid TM system is to find a way so that hardware as well as software TM execution can be monitored with the same low-intrusiveness and without duplicating the event logging mechanisms for each execution mode. We present the design of the event logging architecture in the next paragraph.

5.2.1 Design of the Event Logging Extensions

The design of the event logging extension for the TMbox system architecture follows the idea of closely tracking the TM application's behavior. Compared to the previously presented STM-approach, the FPGA-based hybrid TM system offers more possibilities to track state changes at a finer granularity without risking to influence the application's behavior. While the STM software solution requires to write the event to a buffer and eventually flush the buffer to disk, the FPGA offers block RAMs to buffer events and transfer these via the secondary ring bus whenever it does not block any other operation. Having a smaller priority for these events compared with regular invalidation traffic assures that the application's behavior is not changed in any way. With the previously designed STM solution prioritizing memory accesses and shifting the flush of the buffer to a preferred point in time is not feasible. Here the FPGA-based architecture offers more freedom. Partly this is also due to the fact that the TMbox architecture features two ring buses with one of them not being used heavily so that the approach takes advantage of the fact that transferring TM events in a best-effort manner over this ring bus poses an elegant solution. As a consequence, the HW-based solution provides a continous stream of events as these are generated whereas the STM solution rather features transfers in bursts (depending on the size of the buffer).

Figure 5.9 displays a TMbox system with 8 cores that is enhanced with logic to log events. In order to preserve the events for a later post-processing step, the connection of the FPGA to the host computer gains importance. Over the PCI-Express channel, that connects the host computer with the FPGA, the host polls the memory area that holds the TM

Message header		Message data		
2 bits	4 bits	20 bits	4 bits	4 bits
Message type	CPU sender ID	δ-encoded timestamp	Event type	Event data

Table 5.3: Format of the events that are transferred as packets (cf. to [189, 111]).

events. The host copies these events and writes them to trace files. This step preserves the (otherwise lost) events after the run on the FPGA and enables further post-processing steps such as behavioral analysis or visualization on the host. Section 6.2.4 holds an example of a successful optimization through tuning the parameters guided by a post-processing step.

5.2.2 Implementation Details

The presented approach records the TM application's behavior through tracking fine grain state changes in the TM hardware [111]. As a prerequisite to tracking these state changes, we need some means to detect state changes first. Figure 3 in [190] illustrates the finite state machine of the cache. Specific actions e.g., commit or abort trigger state transitions. In the extended version for event logging, these transitions trigger the generation of a corresponding event. In Figure 5.9, the newly introduced component for event logging is called *Event Generation* and is shown to extend the Honeycomb processor core. As already mentioned, the extension is rather made to the finite state machine of the cache that is closely coupled with the processor. The event generation unit passes a generated event to the *Log Unit*. The Log Unit manages a first in, first out (FIFO) queue with 32 entries for events. This FIFO preserves the order of events so that there is no need to reorder events during post-processing. The Log Unit equips each event with a time stamp prior to placing it in the FIFO. The FIFO queue buffers events until a free slot on the secondary ring bus is available to transfer the event to the Bus Controller. The Bus Controller places the event in the added PCIe FIFO queue. Experiments reveal that having 32 entries in the FIFO of the Log Unit is sufficient to correctly handle a hand-written stress test. Even for this test, in which 8 cores generate events at a high rate, the occupation of the FIFO did not exceed 4 entries. This shows that the chosen parameters are well-suited to handle phases of high load correctly.

Table 5.3 shows a detailed view of the trace format. The secondary ring bus defines the trace format so that arbitrary messages are not possible. The message header holds a message type with 2 bits and a CPU sender ID with 4 bits. The message type distinguishes invalidations and events. The CPU sender ID may distinguish up to 16 CPUs. The message data contains a timestamp with 20 bits, an event type with 4 bits and 4 bits event data. The timestamp is δ-encoded, meaning that only the time difference to the previous event is recorded. This space-efficient coding scheme enables a cycle-accurate monitoring of events. The downside is that a timespan between two logged events must not exceed 2^{20} cycles. Otherwise the synchronization of the events is lost and the reconstructed application's behavior in the post-processing step would not match the genuine application's behavior. On the TMbox architecture, the processor has a core frequency of 50 MHz, the time span equals 20 ms. Thus, for our use cases with high transactional activity, this solution suffices. A possible enhancement would be to generate a keep-alive message prior to the timespan being exceeded. Then, the next event would either be relative to this artificially generated message or generate the next keep-alive message if the time span is exceeded again. This would be a possible solution for this issue. Monitoring the pure STM execution of TMbox

Event Logging Type	Area Overhead (per CPU core)	Actions Tracked
STM-only (x86 host)	NONE	SW start tx, SW commit tx, SW abort tx
STM-only (FPGA)	32 5-LUTs + 1 BRAM	SW start tx, SW commit tx, SW abort tx
HTM-only	129 5-LUTs + 1 BRAM	HW start tx, HW commit tx, HW abort tx,lock bus, unlock bus, HW inv, HW tx r/w, HW PC
HyTM CG	129 5-LUTs + 1 BRAM	HTM-only + STM-only
HyTM FG1	129 5-LUTs + 1 BRAM	HyTM CG + SW tx r/w + tx ID
HyTM FG2	129 5-LUTs + 1 BRAM	HyTM FG1 + SW inv + SW PC

Table 5.4: Area overhead per processor core and the tracked events in different tracing configurations – taken from [189].

also poses a challenge because so far the monitoring hardware only tracks changes of the HTM state. Due to the flexibility of the FPGA and using a soft processor core, it is possible to add an instruction that only generates a corresponding event. These additional event logging instructions for the STM part are inserted by the compiler/assembler. Thus, the user of the architecture does not have to insert those. The other advantage is that the event generating and logging mechanisms are utilized in the same way as with the HTM states. Thus, the run time overhead of profiling the software execution of a hybrid TM system consists of one additional instruction per generated event. The following paragraph compares the overheads of the different event logging systems at different logging granularities.

5.3 Comparison of SW- and HW-based Monitoring of TM Events

This paragraph compares the different tracing variants in terms of area and run time overhead. Table 5.4 presents the overhead of the proposed event logging mechanism in terms of area overhead with respect to the number of actions that can be tracked. *STM-only (x86-host)* is the binary tracing variant from Section 5.1 with a buffer size of 100 K elements per thread executed on ExpX5670. Of course this tracing variant does not come with any overhead in terms of hardware. It tracks start, commit and abort of transactions in software. A comparable setup, *STM-only (FPGA)* with the event logging mechanisms on the FPGA architecture requires 32 lookup tables and one block RAM additionally on the BEE3 platform using one of four LX155T Xilinx FPGAs at the UPC/BSC. For an *HTM-only* setup that monitors the same actions as the previous setups only in hardware and additionally tracks bus events, reads, writes, invalidations and the program counter, the area overhead is 129 lookup tables and one BRAM. This is the maximum area overhead achieved in this experiments that is also matched by the three remaining hybrid designs. The three hybrid variants are: coarse grain (*HyTM CG*) that tracks HTM-only and STM-only events. *HyTM FG1*, additionally to the events of *HyTM CG*, also handles reads, writes and the transaction id. *HyTM FG2* adds invalidations and the program counter to the events of *HyTM FG1*.

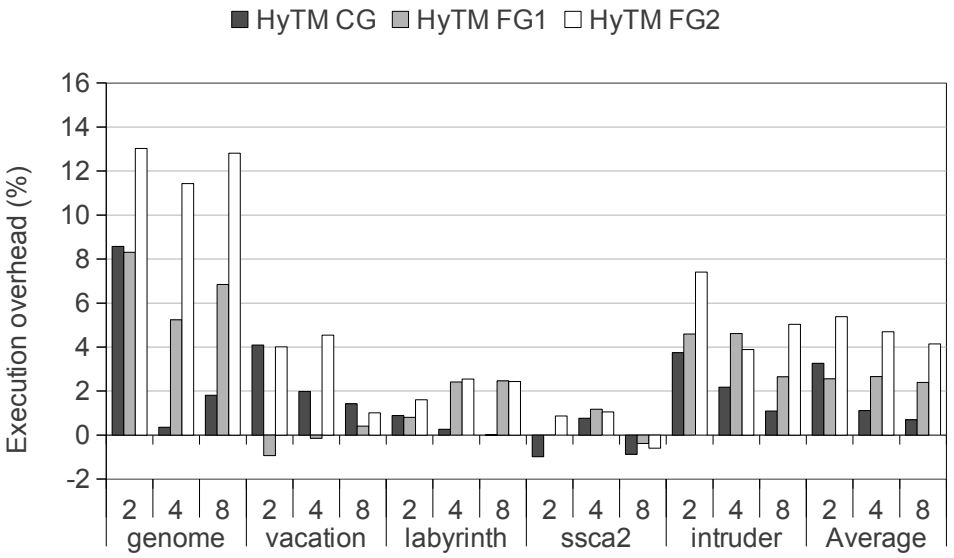


Figure 5.10: Run time overhead in % for different Hybrid TM tracing levels, core counts and applications. Each bar is an average over 20 runs – taken from [189].

Figure 5.10 compares the run time overhead for the three hybrid TM event logging granularities using the EigenBench [94] mimicry of the STAMP benchmarks [24]. The run times from logging events while tracing the application characteristics of `genome`, `vacation`, `labyrinth`, `ssca2` and `intruder` provide a thorough picture. Apart from two outliers – `vacation` with 2 threads and `ssca2` occasionally – with both showing a small speedup, the general trend is as expected: the tracing variants that log more event types, also show the higher influence. E.g., HyTM FG2 has the highest run time overhead between $4\,\%$ and $6\,\%$ on average depending on the number of cores. Moreover, `genome` has the highest influence with HyTM FG2 with a $\approx 13\,\%$ longer execution time. The small speedups may be explained by the impact of small delays on transactional execution: in case a threads is delayed because a TM event must be generated, it may not run at the exact same time as without tracing enabled and, hence, may not generate a conflict with another transaction. This reduces the number of rollbacks and may cause a small speedup. As the performance numbers confirm, this trend is not very pronounced and reduces the execution time by $\approx 1\,\%$.

Figure 5.11 finally compares the execution overhead in run time of *STM x86* and *STM FPGA*. The experiments use 2, 4 and 8 cores and again the Eigenbench variants of `genome`, `vacation`, `labyrinth`, `ssca2` and `intruder`. The range of application's behaviors demonstrates a varying influence. For `genome` with 2 cores, STM x86 is only influenced by $3.5\,\%$ whereas STM FPGA is close to $\approx 8\,\%$. For `genome` with 4 and 8 cores both variants are similar with slight advantages for STM FPGA. On the contrary, `ssca2` shows almost no influence with STM FPGA or even a small speedup and yields $\approx 10\,\%$ overhead on average across core counts. STM x86 extends the run time of `intruder` with 8 cores by $> 14\,\%$. This is the highest influence measured in this experiment. We observe the following overall trend for STM x86 a higher core counts also yields a higher influence of the run time. For STM FPGA this trends does not hold. The highest average influence is with 2 cores for STM FPGA. Moreover, STM FPGA yields the lower influence on average due to the additional FPGA hardware and the clever design of the tracing mechanisms.

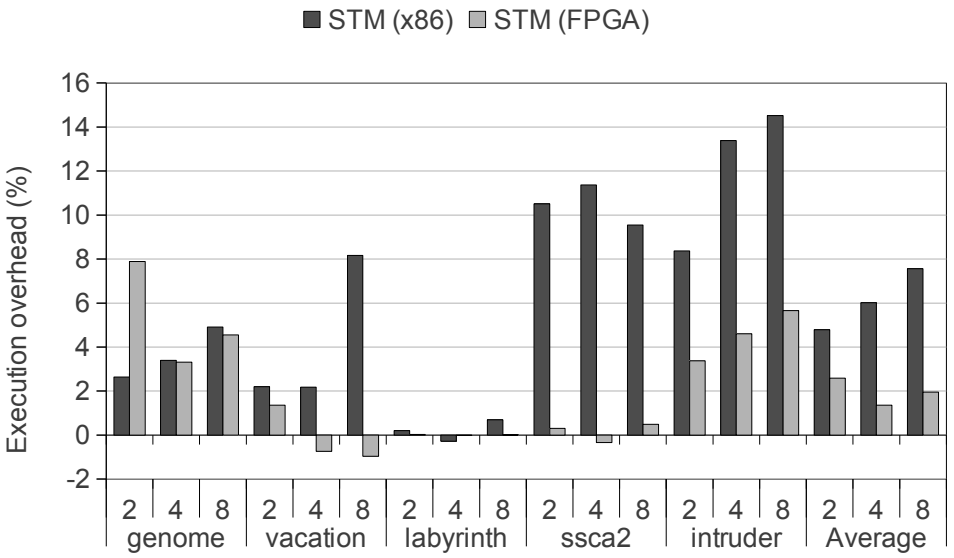


Figure 5.11: Runtime overhead in % for STM-only (x86-host) vs. STM-only (FPGA) with different core counts and applications. All bars represent averages over 20 runs – taken from [189].

The tracing of execution in HTM mode yields lower overheads than that of the STM mode due to the additonal instructions required. Hence, applications that mostly execute in HTM mode experience less overhead through the tracing machinery than applications that heavily use the STM mode. To summarize, the average influence for the considered benchmarks of $< 8\,\%$ for STM x86 is also a respectable result, especially when considering that there is no additional hardware to assist the tracing on the x86 architecture and, thus, this approach can be ported to any commodity architecture.

5.4 Summarizing the Trace Generation

In this chapter, we present two TM-specific solutions for capturing and preserving the TM application's behavior for STM on an x86 architecture and hybrid TM on an FPGA.

For the STM solution, TM events are logged, buffered and compressed inside a word-based Software Transactional Memory library. This approach substantially increases the throughput and reduces the application disturbance in comparison with a state-of-the-art binary translation tool (Pin). The more sophisticated trace generation variants employ compression algorithms to reduce the amount of data to be written. The ZLIB and the LZO compression schemes are compared with non-compressing variants. The results show that especially adding dedicated compression threads does have benefits: for large data sets the influence on the runtime is reduced significantly. The trace data is compressed with a factor of up to 10. The compression ratio of the ZLIB algorithm is superior to LZO, but also leads to an increased runtime for large data sets. Further, LZO has a small influence on the application behavior compared with ZLIB due to its multi-threaded trace compression implementation. For capturing the genuine TM application's behavior this low-intrusiveness is the most important advantage of the developed LZO scheme. Thus, retrieving information throughout this thesis is either done with LZO compression and

minimum of 2 compression threads or without compression and a buffer size of 100K or 1M elements.

The presented FPGA-based tracing approach augments the hybrid TM system (TMbox). An Event Generation and a Log Unit extend each processor core in order to track changes of the TM state. Each state change of interest triggers the generation of a corresponding event through the Event Generation unit. The Log Unit receives the event, equips it with a time stamp and places it in a FIFO. The event is then transferred during the idle times on a secondary ring bus in order to minimize the intrusiveness. To also monitor software execution of the hybrid TM system, an additional instruction triggers the generation of a log event. This event takes the usual route through the hardware. This approach achieves a continuous stream of events that resembles the application's behavior. The additional hardware requirements are modest and the run time overhead is limited to one additional instruction to monitor the software execution (per event).

A comparison between tracing the software execution of transactions on the FPGA platform and the x86 host deepens the understanding of the techniques. A general and expected trend is that the overall influence on the run time is lower for the hardware-assisted tracing with STM FPGA. As a result some benchmarks show a low influence with STM FPGA-based tracing and a high influence with STM x86-based tracing (e.g., `ssca2` and `intruder`). `genome` with 2 threads also shows a higher influence with STM FPGA compared to STM x86. The reason for this behavior is that transactions in `genome` exceed the hardware capacity during the hybrid TM execution and revert to the slower STM execution with higher overheads. In general using the STM FPGA machinery for generating traces is preferable because of its lower run time overhead and lower intrusiveness. In cases where an architecture with an FPGA is not available, the STM x86 approach has also been shown to have a low influence on the run time and comes with the advantage of also being portable to architectures without FPGA extensions.

6. Visualization and Tool Support for TM Applications in Unmanaged Languages

This chapter presents a framework for the optimization of TM applications written in unmanaged languages, e.g., `C`. Section 6.1 presents the design and implementation of the toolchain that extracts and processes run time information of the STM. Section 6.2 shows how we use this information to uncover optimization potential. We integrate the optimization of applications with hybrid TM, as described in [189], in Section 6.2.4. Section 6.3 demonstrates how to apply TM to the method of Conjugate Gradients and evaluate the performance in detail using hardware performance counters as described in [92, 106, 176]. Phase detection algorithms for TM applications are presented in Section 6.4 and the EigenOpt approach in Section 6.5. Section 6.6 concludes this chapter.

6.1 A Toolchain for the Optimization Cycle of TM Applications

Tool support may help the application developer to discover and avoid bottlenecks in TM applications. Researchers proposed to assist the programmer through visualizing the transactional memory application's behavior [220, 127, 123]. These works are the closest related works and a discussion that contrasts each of these works with our work is presented in Section 3.3. Since our goals is a tool environment for applications written in `C` or `C++` which are unmanaged languages, a lot of the techniques presented in [220] can not be transferred directly because they rely on e.g., the garbage collector. Hence, we research how to obtain similar information and improve on [127] through filling in the gaps. [123] uses `Java` and sampling techniques tailored for the Solaris OS. We aim at a broader applicability and portability of our tools that additionally include the readings of hardware performance counters. Thus, we believe that the presented tool environment works for every language that supports calling functions written in the `C` programming language and runs on architectures that TinySTM can be ported to. Moreover, we focus our optimization efforts on practical aspects e.g., the transaction size and present a programmer centric and systematic approach to guide the programmer to an efficient TM application.

This chapter introduces a framework for the Visualization and Optimization of TM Applications (VisOTMA). The framework and its components specifically targets applications written in unmanaged languages such as `C` or `C++`. With VisOTMA the programmer benefits from the visualization of the run time behavior and receives additional support to design, rate, and optimize the transaction layout. The key feature of VisOTMA is the visualization of the application's behavior that uncovers bottlenecks (e.g., frequently conflicting transactions) or pathological execution patterns. The visualization is of great importance for experienced as well as untrained programmers to identify pathological execution patterns. In the absence of these patterns, our framework additionally supports the optimization of the application through refining the transaction size. Traditionally, the transaction size is set intuitively by the programmer and is later refined through an trial-and-error process. The programmer faces the following dilemma. On the one hand, the transaction should be as long as possible. This exploits the optimistic concurrency of transactions by leveraging the additional parallelism over lock-based approaches and amortizes the costs for setting up a transaction through the available parallelism. On the other hand, the costs for aborting a transaction should be small. Thus, the number of memory locations which are affected by a rollback of the transaction must be small. As a consequence the transaction size should be small. Without further knowledge the application developer would have to implement two versions of the application: one with shorter, one with longer transactions. Benchmarking these gives him or her the confidence that he or she will choose the right one for production use. Our approach builds on a reference application that correlates the conflict potential of the application and the transaction size. Through profiling this reference application, a simple metric is established that helps to direct the transaction length of the application that should be optimized. Moreover the metric could also be used when designing a TM application so that the transaction size as a function of contention becomes a separate constraint in the design process.

Our goal is a comprehensive toolchain for the Visualization and Optimization of TM Applications (short VisOTMA). VisOTMA is supposed to support the programmer in designing, rating and optimizing a TM application. Figure 6.1 presents an overview with all components. The visualization shall highlight bottlenecks in the TM application. Therefore, static (left hand side) and dynamic/run time information (middle) must be collected and matched. To preserve this information, we employ static and dynamic instrumentation and save the data in event logs. A post-processing step then finds dependencies between events. These are likely to become bottlenecks at run time. A correlation of this data with the source code of the application helps the programmer. The inexperienced programmer lacks the knowledge of TM programming that an experienced programmer already acquired. In our framework a metric (right hand side) substitutes this knowledge concerning the transaction size and guides the untrained programmer. Some theoretical thoughts on the performance-critical parameters of a transaction are the foundation for an example reference TM application, called Parameterizable Synthetic TM Application (*PSTMA*). Experimental results with this application show a profitable transaction length as a function of contention. This application determines for each contention level the best size of a transaction provided that other parameters of the TM application, e.g., synchronization to computation ratio, match. In case of large discrepancies, another benchmark, e.g., CLOMP-TM, could substitute PSTMA. An algorithm for transforming these findings into an optimized transaction layout follows. By calculating a simple metric, the untrained programmer can exploit this knowledge to tune the transaction size.

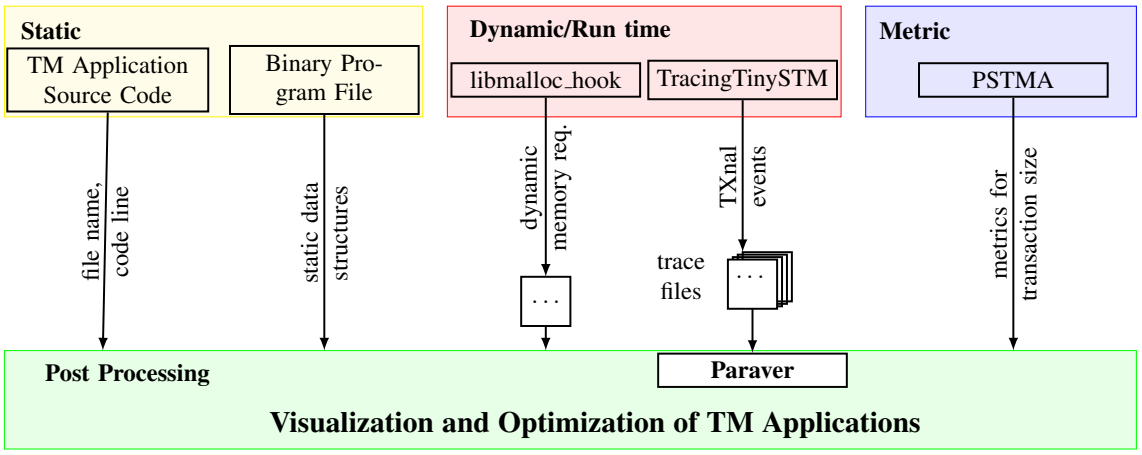


Figure 6.1: Components and interplay in the TM Visualization and Optimization framework.

6.1.1 Studying the Influence of Transaction Size on the Performance

The length of a transaction is of major importance when programming with any kind of Transactional Memory system. For hardware transactional memory the programmer has to respect the sizes of buffers and the duration of transactions. Thus, transactions are to be kept relatively small. For hybrid TM systems the same applies, since the hardware part is faster than the software part. For Software Transactional Memory there is no simple scheme to determine the size of a transaction. On the one hand, the transaction should be as long as possible. This exploits the optimistic concurrency of transactions by leveraging the additional parallelism over lock-based approaches. Moreover the costs for setting up a transaction will amortize through the available parallelism. On the other hand, the costs for aborting a transaction should be small. Thus, the number of memory locations which are affected by a rollback of the transaction must be small. As a consequence the transaction size should be minimal. Trapped in this dilemma, the TM programmer neither knows which argumentation to follow. Further, the size of a transaction also depends on the characteristic of the application. Thus, a transaction with high conflict potential favors a different size than one with low conflict potential. The programmer has to take this into account as well.

In the following the crucial performance aspects of transactional execution are formalized with a simple analytical model. From this model, we will derive an example reference TM application. Through introducing the right parameters, this application enables to determine a profitable transaction size experimentally.

Assume a multi-threaded synthetic application which continuously executes transactions over a fixed period of time. The execution time t of a transaction depends on the number of loads (ld), stores (st) and their respective duration (modelled through function t), the contention level (cl) and some variable bookkeeping overheads. The contention level expresses the likeliness of conflicting with a concurrently running transaction. The function ab models the number and duration of actions that are necessary to abort the transaction. This function depends on the number of loads and more importantly stores in that transaction. Further, setting up a transaction and validating on commit are encapsulated in the function penalty (pe) which again depends on the number of loads and stores.

$$t_{txn} = st * t(st) + ld * t(ld) + cl * ab(st, ld) + pe(st, ld)$$

```
__transaction {
  for (i=0; i<iterations; i++) {
    if (i==iterations/2)
      if (rand() < cp)
        stm_store(&cv, tid);
    tmp = stm_load(&array[tid][i]);
    tmp++;
    stm_store(&array[tid][i], tmp);
  }
}
```

Listing 6.1: Algorithmic design of a parameterizable synthetic TM application (PSTMA).

Now, the executed transactions per second ($Txns/s$) of the application depends on the average execution time of all transactions. From this simplistic analytical model, we derive a parameterizable synthetic TM application (PSTMA) that enables us to determine some key parameters experimentally. This has the advantage over using e.g., the EigenBench microbenchmark [94] that a means to steer contention is accessible over a separate parameter. In EigenBench the probability of conflicts depends on a set of variables (W_1, R_1, A_1). Thus, our PSTMA approach is simpler and directly enables to steer contention. In particular, the goal of PSTMA is to establish a correlation between the contention level cl, the transaction size ($ld + st$) with regard to the amount of executed transactions (Txn/s). Compared with other TM benchmarks described in this thesis such as CLOMP-TM, PSTMA directly reflects the previous model and is even simpler. Consequently the results with PSTMA are not as general as with CLOMP-TM. However, replacing PSTMA with CLOMP-TM in case a TM application does not match PSTMA is a viable option. Listing 6.1 describes the algorithm of the application (PSTMA).

Constructing the application as described in Listing 6.1 has the following advantages: first, the variable generating contention cv and the number of loads and stores are decoupled[1]. Therefore, we can use one application parameter to steer contention (called contention parameter – short cp). The function `rand()` generates a conflict with a parameterized probability. A different design choice would have been to partly overlap the memory regions which are accessed through the loads and stores. However, this would have introduced a secondary parameter that describes whether differently overlapping memory regions influence the application's throughput. Second, the number of loop iterations decides on the length of a transaction. The number of transactional loads and stores are fused in the parameter $iterations$. After processing half of the loads and stores of the transaction, the contention variable (cv) is written. This resembles the occurrence of a conflict after the half of the transactional loads and stores have been carried out. Through this setup, equally many transactional loads and stores are executed before and after a potential conflict is generated. Subsequently, this setup prefers neither early nor late conflict detection schemes because the time that is spent with and without pending conflict are equally long. These simplifying assumptions are necessary to reduce the complexity and find an entry point to a structured optimization process.

Before elaborating on the findings of the execution of the PSTMA, we will sketch how to exploit the acquired knowledge. This PSTMA experiment enables us to detect a beneficial

[1] `array` is indexed with thread id and herewith touches only thread-local memory.

```
/* Independent of ATO */
(1) determine x_max where Tp_PSTMA is maximal.
(2) find intervals such that
∀x ∈ [x_1, x_2] : Tp_PSTMA(x) > (1 − p) * Tp_PSTMA(x_max)
where p ∈ [0, 1] represents the acceptable distance from the Optimum.

/* ATO specific steps */
(3) determine cl_ATO, st_ATO, ld_ATO experimentally
(4) if ((st_ATO + ld_ATO) ∈ [x1, x2])
    /* ATO is in p−reach of Tp_{max} */
    else {
      /* found optimization potential */
      if ((st_{ATO} + ld_{ATO}) < x1)
        // transactions are to small
      if ((st_{ATO} + ld_{ATO}) > x2) ;
        // transactions are to large
    }
```

Listing 6.2: Proposed optimization algorithm for TM applications based on PSTMA.

transaction length for a given contention level. The optimization algorithm is presented in Listing 6.2 where ATO is short for Application to Optimize. Thus, through determining the contention level of the ATO experimentally, our PSTMA application enables to find a profitable transaction size. The idea is to change the transaction size of the ATO so that it is closer to the optimal value of the PSTMA of the same contention level. An example of applying this algorithm to the transactified `fluidanimate` benchmark is presented in Section 6.2.1. When comparing the presumably optimal transaction size of the PSTMA, with the actual transaction size in the ATO, the programmer may take the following counter measures. In case the transactions size is to small, the following counter measures are suited if they are applicable: fuse two neighboring transactions to one, unroll embracing loop such that two transactions can be fused. If the transactions size is to large, the transaction needs to be split. However, this has an impact on the semantic of the transaction. The resulting two transactions come at the cost of having a different atomicity as the previous one. Thus, the programmer needs to think about the side effects of publishing the intermediate results of the first transaction.

After explaining the theoretical thoughts about the design of the application and how to exploit the results, the findings of the experiments with the PSTMA follow. The throughput in Txn/s is depicted in Figure 6.2. Reported values are averages over 30 runs on ExpX5670 to account for variations in execution times. For some iterations a noticeable drop in the throughput is observable. These drops correlate with an occasionally higher abort rate which may be due to STM internal increase of the read or write set sizes because these aborts also appear with a contention level of 0.

More importantly, a higher throughput for smaller transactions becomes obvious. This is a logical consequence of the fact that smaller transactions with fewer loads and stores are executed and rolled back faster than longer ones. Further, due to the shorter run times, a conflict with other transactions is less likely to manifest. However, this insight is rather obvious and does not help a programmer when designing a TM application because of the

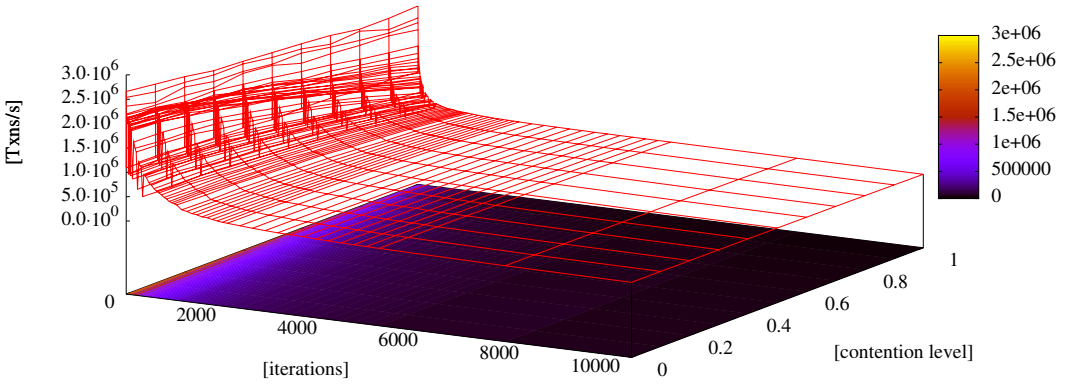
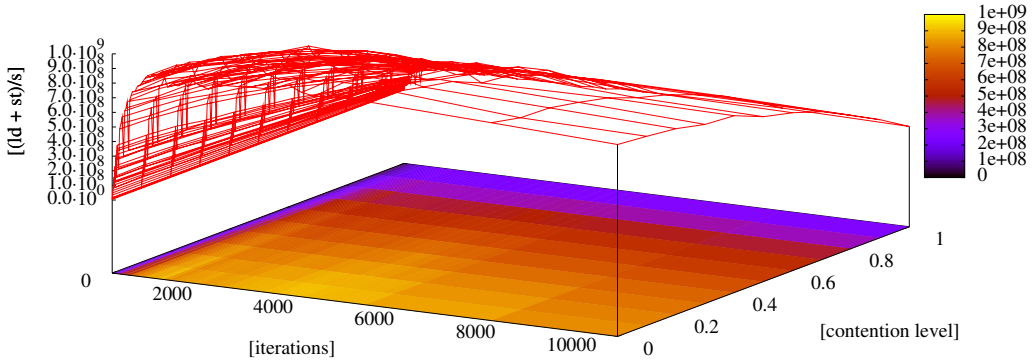


Figure 6.2: Throughput in Txn/s with PSTMA depending on the contention level and the number of loop iterations inside transactions on ExpX5670.

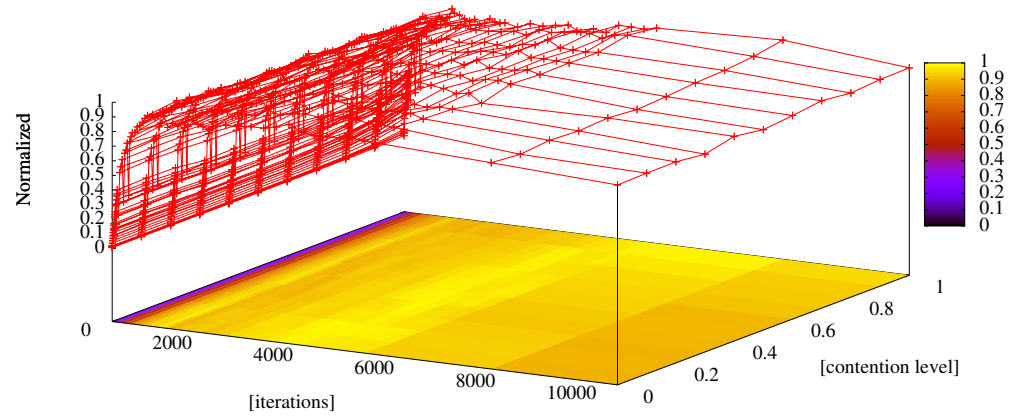
following. Assuming a problem with fixed size, the algorithmic solution requires a fixed amount of load and store operations to shared memory. A programmer needs to know how to encapsulate these loads and stores in transactions such that the resulting program runs efficiently. Therefore, the amount of loads and stores inside a transaction with respect to the achievable throughput in Txn/s needs to be studied. With this information adjusting the transaction layout is feasible. Since the PSTMA runs for a fixed period of time, a throughput of $\frac{ld+st}{s}$ satisfies the before mentioned criteria. Through the construction of the PSTMA, this metric is calculated according to: $\frac{(ld+st)}{s} = \frac{ld+st}{Txn} * \frac{Txns}{s} = 2 * iterations * \frac{Txns}{s}$. Here, the conditional store to the contention variable is omitted for clarity. Figure 6.3(a) shows the results of calculating the $\frac{ld+st}{s}$. A brighter color indicates a higher throughput.

In Figure 6.3(a) the relative difference due to the changes of the contention level are difficult to assess. Thus, we derive a general performance trend that expresses the correlation of the throughput (for loads and stores) and the contention level. Therefore we normalize to the highest achievable throughput for a particular contention level. This is calculated according to: $\forall i : \frac{Tp_{cl_i}(ld,st)}{Tp_{max,cl_i}(ld,st)}$.

The results in Figure 6.3(b) show that the highest throughput per load and store (bright color) depends on the contention level. Without contention the value ranges around $1\,000$ and $4\,000$ iterations and rises to $6\,000$ to $8\,000$ for a contention level of 1. This suggests that a transaction size should contain between $1\,000$ and $8\,000$ loads and stores. These findings attenuate the intuitive explanation that long transactions in a scenario with high contention are not profitable due to the amount of operations that have to be reverted. Nevertheless, it is still wise to avoid high contention through a smart design of the application to avoid wasted work through aborted transactions. For PSTMA the figure shows that long transactions in a scenario with high contention yield the higher throughput of load and store operations. Note that this observation applies to the use of transactions in a simplified scenario as with PSTMA. Here, contention is always generated with the same variable and TinySTM supports that transactions wait for a lock that is taken by another transaction as long as the wait does not cause a circular dependency when using commit-time locking. The results are generated with encounter-time locking and a write-through strategy, but this discussion shows that PSTMA only covers a special case in the processing of transactions that can not be generalized easily. Further, the experimental setup with TinySTM influences the performance and should be identical. In the real world, the programmer will also have the choice to use other synchronization mechanisms to avoid the high costs for rollbacks.



(a) Loads and stores per second as a metric for efficient transactional execution.



(b) Normalized loads and stores to highest throughput in that contention level.

Figure 6.3: Throughput in transactional loads and stores per second. Normalizing these loads and stores reveals whether a given transaction size is profitable for a particular contention level.

Moreover, the memory access pattern with PSTMA is uniform to all threads with just one contention variable so that a more complex memory access pattern should also be considered to mimic the application under test. The use of CLOMP-TM, introduced later in Section 8.3.3, is appropriate in cases where the TM application is not sufficiently modeled through PSTMA or to achieve a comparison with other synchronization mechanims. Often the programmer is not able to apply this knowledge right away, but will be confronted with the task of optimizing a previously written application. Thus, before turning the theoretical thoughts of this section into an optimized TM application, the programmer has to determine the actual application's behavior.

6.1.2 Retrieving TM Events and Memory Requests

The important step before optimizing or visualizing a TM application is to capture its run time behavior. Thus, the application's TM events and memory requests need to be tracked at run time and be preserved. This thesis covers the step of capturing the application's TM events combined with measurements of the intrusiveness of the tracing machinery in Chapter 5. To retrieve the transactional loads, stores, aborts and commits, we rely on event traces generated at run time. These transactional events are matched with the readings of the hardware performance counters with PAPI. This enables to quantify the pressure that

```
# Format:
# pthread_self, size, virtual address, ip of caller, TSC
139758356580096:336:0x2cba010:0x401c93:2828473027711794
...
```

Listing 6.3: Example illustrating trace format for logging dynamic memory requests with `malloc`.

specific STM functions put on particular architectural resources. Further, we employ a technique for logging memory management requests without modifications of the source code. Only the combination of both enables correlating an STM-specific event with an address and a particular data structure which has been allocated dynamically. This is a prerequisite for a comprehensive optimization of the application.

Statically Allocated Memory

The interesting statically allocated memory comprises i.e. global variables and small arrays. The memory addresses are assigned by the linker. Thus, the program binary already contains all necessary information. For extracting these information, the Linux environment provides a tool called `objdump`. However, other operating systems provide similar tools. Thus, this does not prevent the adoption of the presented techniques for other operating systems. `Objdump` provides the name of the variable and the corresponding address when the application is compiled with debug symbols. Other tools (in our case `grep`) process the output to search for a particular address from the TM trace files. Herewith the connection between the statically allocated memory and the TM event traces is established.

Capturing Dynamic Memory Requests

In order to obtain a complete view of the memory accesses of a TM application, accesses to dynamically allocated memory must be tracked as well. For capturing these dynamic memory requests, we rely on a library called `libmalloc_hook`. Tao et al. proposed this library in [199]. More specifically, this library intercepts calls to `malloc` and `free` by means of the `LD_PRELOAD` mechanism. This very same mechanism has been exploited for dynamic instrumentation purposes in the context of the DynamoRIO project [22]. This mechanism enables to call a self-made library prior to calling the original one. Thus, we extract the following information for each call to `malloc`: the thread identifier returned by `pthread_self`, the size of the allocated memory in bytes, the virtual start address, the value of the time stamp/cycle counter, and the instruction pointer of the caller. Listing 6.3 illustrates an example of an invocation of `malloc` and the trace format. For `free` this is reduced to the virtual address, the time stamp counter (TSC), and the instruction pointer of the caller as is illustrated in Listing 6.4.

We correlate the obtained information with other information about TM events as follows. With the instruction pointer of the caller, the code location of a dynamic memory request is correlated with the file and line number by using the `addr2line` tool that relies on debugging symbols in the binary. The file and line number are required to identify the name of the data structure related to the recorded virtual address. The time stamp counter enables to merge the memory event traces with the traces of the TM events and establish a correct ordering of events for each thread. This is the prerequisite to incorporate

```
# Format:
# pthread_self, virtual address, instruction pointer, TSC
139758356580096:0x2cba010:0x402907:2828473087443556
...
```

Listing 6.4: Example illustrating trace format for logging dynamic memory requests with `free`.

memory management information in the visualization process. Moreover, the information is also required to generate overall statistics on the memory allocated and freed during a program run, identify the point in time when it happened and, thus, reveal an inefficient use of memory management routines of the application. The library `libmalloc_hook` also records calls from the STM and other system libraries. This additionally enables investigating the memory requests of the STM library under the condition that these are also compiled with debugging symbols.

6.1.3 Visualization with Paraver

Paraver is a visualization tool that is developed at the Barcelona Supercomputing Center (BSC) former at the CEPBA-UPC. The power of Paraver has been demonstrated in [30] where the Sweep3D benchmark was optimized. Further, Paraver has been extended with an interface to the CAPO tool. CAPO supports the programmer to parallelize the code [107]. However, Paraver is powerful and many more applications of it can be found.

For the visualization of the transactional memory application behavior, the following attributes were the dominant ones: customizable semantic of events, visualization of large event traces, the possibility to build derived metrics, and the layered approach. The latter enables to write filters that enable/disable the visualization of selected events. This helps to focus the programmer's attention (e.g., on communication patterns). With transactional memory the communication is implicit through reading or writing shared memory locations. In contrast to explicit communication through dedicated primitives (e.g., in the context of MPI), these implicit communication patterns need to be extracted from the TM event traces. The necessary transformations and the extraction of communication patterns is described in the following two paragraphs.

Transformation of Data

The TM event trace files contain an event log of all transactional events of a specific thread. Thus, all events happened in-order and are recorded accordingly. Specific events (e.g., start, commit, abort) are logged together with a time stamp (TSC). Each thread writes to a separate trace file. Paraver visualizes one trace file that contains the information of all threads. Thus, a ParaverConvert tool needs to merge and process these trace files. Further, the two formats do not compose canonically because Paraver distinguishes between states and events. States are defined through a duration. Consequently states have a start and an end time stamp. Therefore, the TM events need to be mapped to the states and events of Paraver. Mapping all traced TM events onto Paraver events would lead to a non-informative visualization because events are displayed as small flags in the horizontal time line view. Thus, we decided to fuse the start and the outcome of a transaction in one state. The state itself depends on the outcome of the transaction. Thus, only commit and abort states

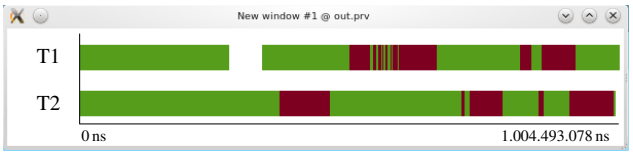


Figure 6.4: Example of a TM application visualized with Paraver. Bank runs with 2 threads on ExpX5670.

```
set
get_addr_set (i,j) {
  // determine overlap of transactions
  if (S_i >= S_j)
    if (S_i < E_j)
      return conf_addr_set(i,j);
  if (S_j >= S_i)
    if (S_j < E_i)
      return conf_addr_set(i,j);
  return ∅
}

set
conf_addr_set (i,j) {
  set ret = ∅;
  ∀a ∈ L_i ∃b ∈ W_j : (a = b) ⇒ ret = ret ∪ a;
  ∀a ∈ L_j ∃b ∈ W_i : (a = b) ⇒ ret = ret ∪ a;
  ∀a ∈ W_i ∃b ∈ W_j : (a = b) ⇒ ret = ret ∪ a;
  return ret;
}
```

Listing 6.5: Detecting communication patterns between transactions.

exist. These are displayed as fully colored bars. ParaverConvert maps transactional load and store events to newly defined Paraver TM load and store events. Figure 6.4 shows an example of a TM application (a bank application) run with 2 threads on ExpX5670 that is visualized with Paraver. On the y-axis two threads are displayed. Both threads are executing transactions concurrently (apart from the gap in `T1`). The concurrent execution leads to conflicts between the threads and these lead to an abort of transactions. Aborted transactions are colored red. Transactions that committed successfully are green.

Detecting Contention in Transactions

In the following we present the algorithm to extract the communication patterns between transactions that has been implemented in the ParaverConvert tool. Let S_i be the start point of txn_i and E_i be the end point of txn_i. E manifests either as commit or as abort event. Let L_i be the set of memory addresses touched by a stm_load and W_i represents the set of memory addresses of a stm_write in txn_i. Let C_{ij} be the set of conflicting addresses of txn_i and txn_j. The algorithm to detect conflict potential between two transactions looks as indicated in Listing 6.5. First, the time stamps are checked whether both transactions overlap. Otherwise an empty set is returned. If the transactions overlap, the read and write

Figure 6.5: Dependencies between transactional accesses of two threads visualized with Paraver.

sets are compared whether a conflict may occur. This requires to check the read set of txn_i against the write set of txn_j and vice versa. As a last step the write sets of both transactions are compared. The algorithm is similar to the one presented in [220]. Moreover, this algorithm also considers transactions that overlap partially and did not actually conflict during the traced run. This is enhanced by adding a sliding window that replaces the check for the overlapping transactions. When setting the sliding window to two times the size of the longest transaction, the amount of compared transactions increases significantly. Thus, the algorithm is capable of accounting for some amount of variability during the execution and is more general than previous ones.

By regarding all addresses in the read and write sets, the programmer does not first fix the address that is responsible for a specific conflict only to find out that the next one needs to be fixed as well [220]. Also this point has been made previously in the context of a managed language, it holds for our case as well. When regarding all addresses that may lead to a conflict the programmer can revise the memory layout to avoid these conflicts. For the integration with Paraver, the pass through the event traces emits the discovered dependencies as communication events. The affected addresses and their points in time are encoded in this event. These events are derived from the Paraver MPI send and receive events. Similar to them, the newly generated events express the dependency between two transactions similar to expressing the dependency between two MPI tasks. However, extracting these communication events from the trace files is more cumbersome due to the implicit communication between the threads.

The communication patterns in transactions can now be visualized with Paraver. Figure 6.5 shows an example TM application with the Paraver filter for communication events enabled. In particular yellow lines connect the transactions where an address may cause a conflict between these transactions. When clicking on one of the involved transactions, information about the events of that transaction are shown in a separate window. Thus, the timely occurrence of these events as well as the memory address are listed. This information is correlated with the dynamically and statically allocated data structures. Then, the programmer knows which data causes contention and may adjust the memory layout.

Statistics on Transactional Execution

Of great importance for the programmer is a well-defined set of statistics enabling to find the hot spots. In our particular case the generation of statistics is integrated in the transcoding of the TM event traces. This has the advantage that the events are read anyway and accumulating a few counters when encountering an event does not cost much. Thus, the statistics come essentially for free during the post processing step. Of great importance are overview statistics that reveal the number of transactional events executed in total. These give a first impression whether the program behaves as expected. If the number of executed transactions is high and the abort rate very low it will probably not

```
Global Statistics:
Transactions: 951 Abort: 272 (0.286) Commit: 679 (0.714)
Commit: Loads  =   679000 Stores = 679438
Abort:  Loads  =   136000 Stores = 136272
Time:   Commit = 97679447 (0.830)
        Abort  = 19976802 (0.170)
```

Listing 6.6: Global statistics on transactional execution with number of aborts, commits and impact on the global execution of the application.

pay off to optimize the application. More important for the individual optimizations are the thread specific statistics. The global thread view summarizes the transactional events of the threads. If one thread executes substantially more transactions than another one this indicates a load balancing problem. In case one threads experiences significantly more aborts than others, the relationship between transactions (who aborts whom) should be investigated. Listing 6.6 holds an example of the global statistics. 951 transaction have been attempted. Other publications such as [123] refer to this as physical transactions. From these 951 attempts 679 committed successfully whereas 272 aborted and have been executed again. The committing transactions carried out 679 000 transactional loads and 679 438 stores whereas the aborting transactions 136 000 loads and 136 272 stores. The Time entry reports the respective time in nanoseconds for the execution of committing and aborting transactions.

The global statistic is important to gain an overview but may hide the fact that one transaction experiences or causes most of the aborts. For closer investigations, a per thread statistic contains the highest level of detail. Here, each transaction is listed with the amount of loads and stores carried out. These are divided into aborts and commits such that it is possible to tell where the most work and time has been wasted. Thus, all loads or stores that had to be undone because of an abort are listed here. This reveals the transaction with the most potential for optimization. A record for an example transaction is shown in Listing 6.7. The local and global values show the share of commits or aborts with respect to all attempts of this transaction and all transactions throughout the program run. For example, aborting this transaction accounts for 14 % of the global transactions but causes only 8 % wasted work. This indicates that other transactions cause more wasted work although aborting less often and, hence, must be larger.

```
Return Address: 40164d
Transactions:   481 (global: 951)
Abort:          136 (global: 0.143 local: 0.283)
Commit:         345 (global: 0.363 local: 0.717)
Abort:  Loads =  68000 Stores =  68136
Commit: Loads = 345000 Stores = 345224
Time:   Abort  =  9792472 (global: 0.083 local: 0.167)
        Commit = 48974513 (global: 0.416 local: 0.833)
```

Listing 6.7: Per transaction statistics with number of aborts, commits and impact on the global execution of the application.

```
# Format: address -> number of conflicts
139749511662944 -> 101
139749511662688 -> 111
139749511662800 -> 117
139749511665392 -> 118
```

Listing 6.8: Sorted list of addresses with number of contentious accesses.

A complementary statistic contains the memory address and the number of conflicts it has been involved in. These statistic is also generated for each thread separately and also distinguishes between read and write accesses. The sorted list reveals the address with the most dependencies (for read or writes of other transactions) to the programmer. An example is shown in Listing 6.8.

This also helps to focus the optimization to the frequently conflicting addresses. The techniques described in Section 6.1.2 and Section 6.1.2 determine the corresponding variable or memory region.

In the programming language `C`, we need to capture the context of the start and the end of a transaction. This is difficult because TM events are logged inside the STM library and the STM API should not change. Inside the library the context, from which the STM function has been called, is lost and we can not work around this by passing additional parameters to the function. Thus, we need to reconstruct the context from where the TM application called the STM library. We solve this by means of the return address. This return address is additionally logged at transactional events such as start, abort, commit through an inline assembly instruction. With the return address of the begin of the transaction the following loads and stores can be assigned to that particular transaction. Moreover, the tool `addr2line` enables to map the return address to the code line of the source code. Thus, a full reconstruction of the executed transaction from the TM event traces is possible. However, a similar lightweight solution for the complete call stack is still under construction. In terms of portability and usability, the best candidate so far is the `backtrace` function implemented in `libc` but this adds significant overhead compared to a single assembly instruction that we currently use.

As an important feature the VisOTMA framework integrates the reading of hardware performance counters. The PAPI interface is used to read performance counters at a transaction granularity as also shown in Section 5.1.1. The TracingTinySTM sets the type of the PAPI events of interest to the programmer at compile time. These events are read on transaction start, commit, or abort emitted as separate event. Hence, the difference between the commit or abort and start values correlates with the amount of events generated through the execution of the transaction. These events help the programmer to determine the effective amount of wasted work when an transaction aborts more precisely. So far the wasted work is estimated based on the size of the read and write sets. This is a good indicator but does not cover expensive floating point operations that have been carried out on the loaded data. Hence, our approach improves upon the state-of-the-art through enabling the programmer to measure the amount and type of instructions that have to be undone. E.g. when measuring instructions retired with the PAPI counters, these event counts also include the overheads introduced through the use of an STM system and, hence, represent the true amount of work that has been wasted in case the transaction aborts and not just the instructions present in the transaction.

cp	ApT	[min,max]	Txn size
0.0	0.097	$[0.079, 0.115]$	3 500
0.1	0.122	$[0.101, 0.148]$	3 750
0.2	0.182	$[0.160, 0.208]$	3 250
0.3	0.276	$[0.251, 0.315]$	4 000
0.4	0.402	$[0.376, 0.448]$	3 750
0.5	0.575	$[0.540, 0.630]$	1 000
0.6	0.814	$[0.770, 0.895]$	3 750
0.7	1.154	$[1.093, 1.262]$	4 000
0.8	1.700	$[1.603, 1.898]$	4 000
0.9	2.677	$[2.519, 2.921]$	5 000
1.0	5.144	$[4.843, 5.665]$	7 500

Table 6.1: Relation of contention parameter and ApT.

6.2 Revealing Optimization Potential

The STM library, we are using throughout the experiments is TinySTM [62]. In the following, we demonstrate how we transactify and optimize a real-world application from the PARSEC benchmarks suite with the VisOTMA framework. Here, the focus is on optimizing the size of the transaction. Then, we demonstrate the usefulness of the visualization component when detecting pathological TM execution patterns. In particular the StarvingElder and FriendlyFire pattern reveal the strength of the visualization in highlighting the bottlenecks.

For our experiments all TM applications run alone and consecutively on the Westmere system ExpX5670 (cf. to Section 4.3). The tracing functionality is implemented inside the TinySTM (version 0.9.9) [62] as described in Section 5.1. All reported execution times are averages over 30 runs to compensate for variations in the execution.

6.2.1 Transaction Size

Comparing the transaction length of the PSTMA and a transactional benchmarks is straightforward. A comparison of the read and written memory locations of the respective applications yields the desired results. When identifying the contention level of an application, the course of action is not as obvious. The PSTMA comes with the parameter cp to generate contention. However, this can not be compared directly to the statistics gathered from the transactional execution of a STAMP application [24]. In these cases contention is not generated artificially. Thus, a metric that expresses the level of contention more meaningful is desired. Therefore, we relate the abort rate ($Aborts/s$) where s is short for seconds to the throughput in transactions ($Txns/s$) such that $Aborts/Txn$, as from now called ApT results. This reflects how many aborts are carried out on average for a committed transaction. This metric is derived from the trace data of PSTMA and STAMP equally well. Further, the metric reflects the contention level since contention leads to aborted transactions.

While PSTMA has specific parameters to define the TM characteristic, real world TM applications are defined through other problem related parameters. As a consequence the programmer needs to match the measured transactional characteristics with the ones from the PSTMA. The metric that is representative and straightforwardly captured for an

```
if (border[index]) {
  __transaction {
    cell.a[j] += acc;
  }
}
else {
  cell.a[j] += acc;
}

if (border[indexNeigh]) {
  __transaction {
    neigh.a[iparNeigh] -= acc;
  }
}
else {
  neigh.a[iparNeigh] -= acc;
}
```

Listing 6.9: Representative code piece of a transactional version of `fluidanimate` with *small Txns*.

application is ApT. Table 6.1 illustrates the mapping of the transactional characteristics of PSTMA to the ApT metric. The left column contains the contention parameter of PSTMA. The corresponding ApT value (which is an average over all runs and iterations) is listed in the middle column. Due to the variation in run time, the ApT value varies. In order to facilitate the matching of application and PSTMA, we also report the min and max values as interval in the right column. This interval serves as entry point for the metric ApT of the TM application. Comparing the ApT value of a program with the intervals enables to rate and find the best assumed transaction size for that application.

In the following, we illustrate the details of this simple optimization as described in Listing 6.2. We transactify and optimize the PARSEC benchmark `fluidanimate` [35, 142]. For a full evaluation and a description of the benchmark, please refer to Section 6.2.3. We start by replacing the fine-grained locks with transactions. Further, STM initialization functions are added and the lock array is removed. This yields our simple first version, in the following called *small Txns*. A representative piece of code is listed in Listing 6.9.

We run the *small Txns* version of `fluidanimate` with the statistical module. This provides simple statistics of this run almost for free: $Txns = 9\,347\,885$, $Aborts = 10\,004$.

Then the $ApT_{small\ Txns} = \frac{10\,004}{9\,347\,885} \approx 0.0011$ is computed. Comparing $ApT_{small\ Txns}$ with the results of the execution of PSTMA (see Table 6.1) yields that the first line with $cp = 0$ matches best. The most profitable transaction sizes for PSTMA with $cp = 0$ is $3\,500$. A comparison with the actual transaction length reveals optimization potential. Thus, the transactions must be enlarged without adding unnecessary instrumentation (also known as over instrumentation).

Optimizing the *small Txns* version of `fluidanimate` is achieved by carefully inspecting the source code. The conditions in Listing 6.9 can be collapsed as is illustrated in Listing 6.10. In the case that both conditions are true, the executed transaction contains twice

```
if (border[index] && border[indexNeigh]) {
  /* long transaction */
  __transaction {
    cell.a[j] += acc;
    neigh.a[iparNeigh] -= acc;
  }

  if (!border[index] && border[indexNeigh]) {
    cell.a[j] += acc;
    __transaction {
      neigh.a[iparNeigh] -= acc;
    }
  }

  if (border[index] && !border[indexNeigh]) {
    __transaction {
      cell.a[j] += acc;
    }
    neigh.a[iparNeigh] -= acc;
  }

  if (!border[index] && !border[indexNeigh]) {
    cell.a[j] += acc;
    neigh.a[iparNeigh] -= acc;
  }
```

Listing 6.10: Transactified `fluidanimate` with the *long Txns* version.

as much transactional accesses as the ones executed before. The name of this optimized version is *long Txns*. The later evaluation (in Section 6.2.3) will show that this optimization not only decreases the run time of the TM version but also increases the scalability for small as well as large input data sets.

6.2.2 Visualization of Pathological TM Cases

In the following the VisOTMA framework highlights pathological TM application behavior. We generate these cases with the EigenBench microbenchmark [94] which is inspired by the pathological behavior described in [17]. We select the two pathological cases to demonstrate the strength of our VisOTMA tool: StarvingElder and FriendlyFire.

StarvingElder – Figure 6.6(a) depicts an overview of a full program run exercising the StarvingElder pattern. Paraver visualizes the concurrently running threads in a time line view. Each transaction executed by a thread is colored according to the outcome of that transaction. A green transaction represents a successful commit whereas a red transaction has been aborted. Thus, the relationship between the transactions in the StarvingElder pattern becomes clear at first sight: `T8` aborts as long as other transactions execute. Additionally, it is almost the only thread experiencing aborts. However, from this view the length of the transactions can not be determined. To see these important details, the programmer simply zooms in and yields Figure 6.6(b). This figure shows that

(a) Overview. (b) Zoomed.

Figure 6.6: Paraver visualizing the StarvingElder pattern generated with the EigenBench microbenchmark.

small transactions repeatedly abort the long running transactions of `T8`. Moreover, small transactions are also aborted occasionally but due to their size, the penalty of a rollback is smaller. In order to investigate the statistics and the load/store events the programmer simply marks the transaction in the visualization. The statistics support the previous impression by showing that the long transaction aborts 2 800 times but contributes only 21 % to the total number of aborts in the program. Thus, aborts of small transactions also account for a large number of aborts. A possible optimization would be to balance the transaction sizes of all threads and make the transactions equally long. This would prevent that one long running transaction is aborted by short running ones, can not make progress due to the many rollbacks, and "starves".

FriendlyFire – The FriendlyFire pattern consists of two (or potentially more transactions) that continuously conflict and abort each other. Figure 6.7 visualizes the communication pattern of this two transactions and reveals the conflicting addresses. By simply clicking the yellow colored dependencies between transactions, the programmer identifies the source of the conflicts. The involved address is then correlated with either statically or dynamically allocated memory and a source code location. Then, appropriate counter measures have to be applied. A solution in this case can be to encapsulate the conflicting addresses in closely nested transactions. Closed nesting supports partial rollbacks of an encapsulated transaction which must be supported by the STM run time system. In case the STM does not support it, the programmer may decide to split the transactions to minimize the penalty on rollbacks or selectively revert to conventional locks for the highly contended addresses. Whichever of the techniques the programmer chooses, each of the presented techniques requires the identification of the source of conflicts that is enabled through VisOTMA.

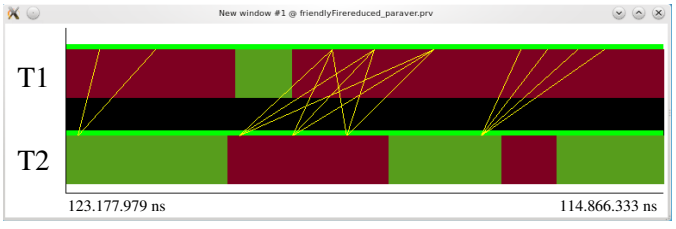


Figure 6.7: Zoomed in dependencies of the FriendlyFire pattern.

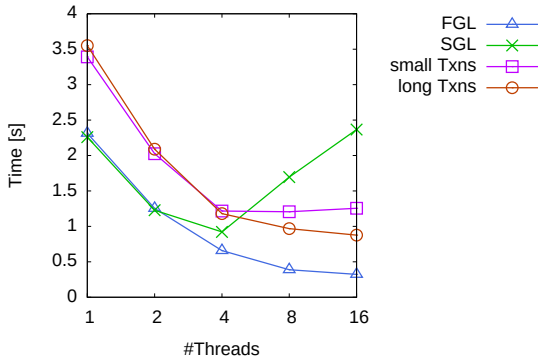
6.2.3 Evaluation of a Transactified PARSEC Benchmark

PARSEC represents the Princeton Application Repository for Shared-Memory Computers that comprises parallel programs suited to rate the performance of multiprocessor machines. PARSEC in version 2.1 comprises 10 applications, 3 kernels, 9 libraries, and 3 tools [35]. From this variety we select the application `fluidanimate`. As the name suggests, it simulates fluid flows with the smoothed particle hydrodynamics method. This method is used to solve the Navier-Stokes equations as described in [142]. The fluid is modelled with particles interacting with each other. Due to the short range of these interactions, it uses a grid with fixed distances. As a result the detection of interacting particles is faster. First, the algorithm computes the density of the fluid at each position of a particle. Second, the acceleration of each particle is computed according to the forces imposed by gravity, fluid pressure, and fluid viscosity. Third, collision detection with a box is performed that results in collision response based on the penalty method. Fourth, the numerical time integration computes the new position and velocity for each particle from the previously calculated acceleration.

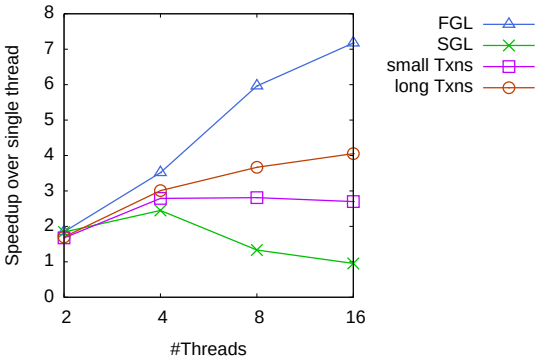
The parallel implementation of `fluidanimate` uses Pthreads and `C++`. Synchronization is originally achieved by means of fine-grained locking (FGL). The grid is partitioned into subgrids of equal size. Each of these is assigned to a thread. Only accesses to adjacent cells at the subgrid boundary need to by synchronized among threads. Thus, these are guarded with fine-grain locks. Updates of particles not in the subgrid boundary cells do not need locks. In addition to the FGL variant, we also implemented a version with one single global lock (SGL) and two TM variants: called *small* and *long Txns*. The *small Txns* replaces all lock and unlock operations of the array of mutexes with a transaction. This results in transaction with one transactional read and one transactional write operation.[2] Listing 6.9 demonstrates a representative code snippet with the *small Txns*. Two if clauses separate the synchronized (in this case transactional) accesses from normal accesses. In the version with *long Txns* the conditions are refactored as shown in Listing 6.10. In case that both conditions are true, one transaction contains twice as much transactional accesses as before.

The performance results of this optimization are significant. Figure 6.8(a) presents the execution time for the *simlarge* input data set and shows that *long Txns* outperforms *small Txns* for 8 and 16 threads. The speedup due to this simple yet effective optimization is 1.30 for 8 threads and 1.43 for 16 threads on ExpX5670. When looking at the speedup over the execution of the respective single thread in Figure 6.8(b), the *long Txns* version again shows a better speedup than *small Txns*. To verify these findings, we execute the same versions with the *native* input data set. Here, the results of the *simlarge* runs are approved: *long Txns* yields a speedup of 1.29 for 8 and 1.39 for 16 threads over *small Txns* (cf. Figure 6.8(c)). This shows as a speedup that slightly increases from 8 to 16 threads for *long Txns* and decreases for small transactions (see Figure 6.8(d)). From a performance perspective the fine-grained locking version is best, but also has the highest complexity for the programmer. For a fair comparison from a programmer's perspective, a single global lock is comparable to TM. For `fluidanimate` both TM variants outperform SGL for 8 and 16 threads. For small thread counts the inherent overhead of Software Transactional Memory (which has been quantified in [27]) dominates the performance. However, the

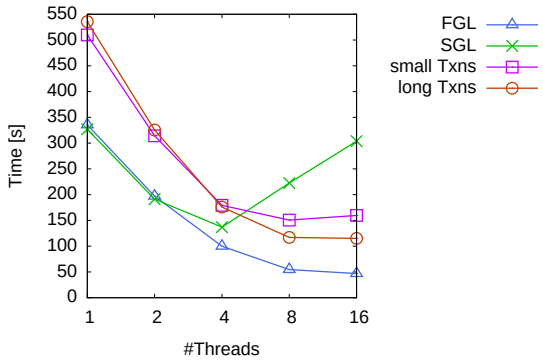
[2] In case the transaction operates on the data type `Vec3`, the transactional loads and stores are carried out separately for each dimension.



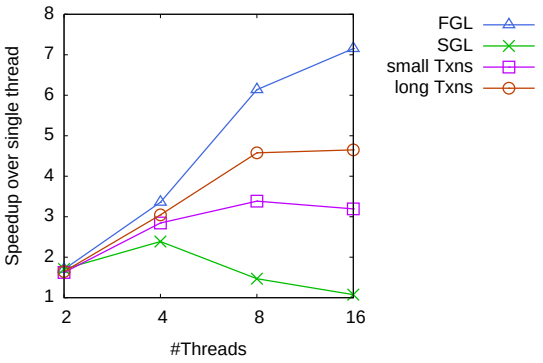
(a) Execution time of `fluidanimate` with input data set *simlarge*.



(b) `fluidanimate` normalized to respective single thread execution time.



(c) Execution time of `fluidanimate` with input data set *native*.



(d) `fluidanimate` normalized to respective single thread execution time.

Figure 6.8: Comparing the execution time of a transactified PARSEC benchmark with *small* and *long* transactions on ExpX5670.

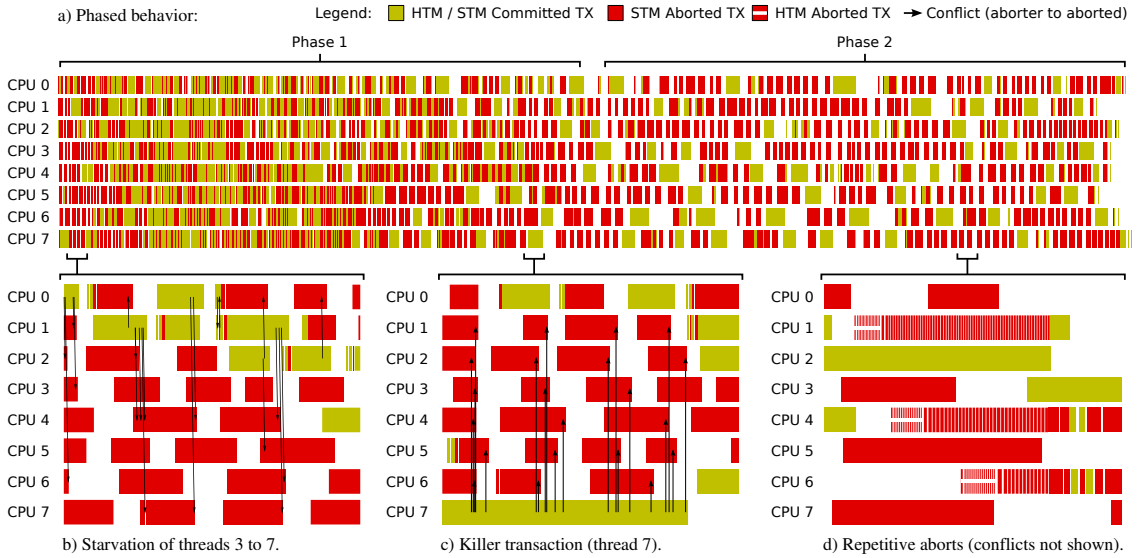


Figure 6.9: `Intruder` benchmark from the STAMP suite [24] executed on the TMbox system with 8 threads. `Intruder` exhibits two execution phases, a starvation pattern involving threads 3, 5, 6, and 7, a killer transaction executed by thread 7 and repetitive aborts. Figure is taken from [189].

proposed TM optimization moving from short to longer transactions while having low contention has been beneficial and shows a far better scalability.

6.2.4 Optimization of Hybrid TM with TMbox

In order to integrate the event traces of the hybrid TM system TMbox in the VisOTMA framework, a simple tool, called *Bus Event Converter*, reconstructs the original order of TM states from the event logs (cf. to Section 5.2). Moreover, *Bus Event Converter* gathers aggregate statistics of the TM execution and emits these and the original TM states as a Paraver trace file to enable visualization of the application's behavior [189]. Especially transitions from hardware to software TM execution are of interest for the performance of the execution with hybrid TM.

Figure 6.9 illustrates the execution of the original `intruder` benchmark, a network intrusion detection application, from the STAMP suite with 8 threads on the TMbox architecture. The visualization reveals two execution phases of the benchmark.

First, all threads assemble the packets for the later detection. This phase is characterized through a higher rate of transactions that corresponds with more transactions being executed in parallel.

During the second phase, `intruder` performs the detection of attacks. The transactions are larger and conflict more often. Further, the visualization also reveals pathological TM executions [17]. The beginning of the execution exhibits a starvation pattern: threads 3, 5, 6 and 7 are repeatedly aborted due to conflicts with threads 1, 2 and 3. This hinders the progress of the repeatedly aborted threads and, hence, is called starvation. Half way through the execution, the visualization uncovers a killer transaction, that is a long running transaction that aborts many shorter transactions. These pathological executions are embedded in the overarching execution of the two phases and, hence, do not dominate the execution.

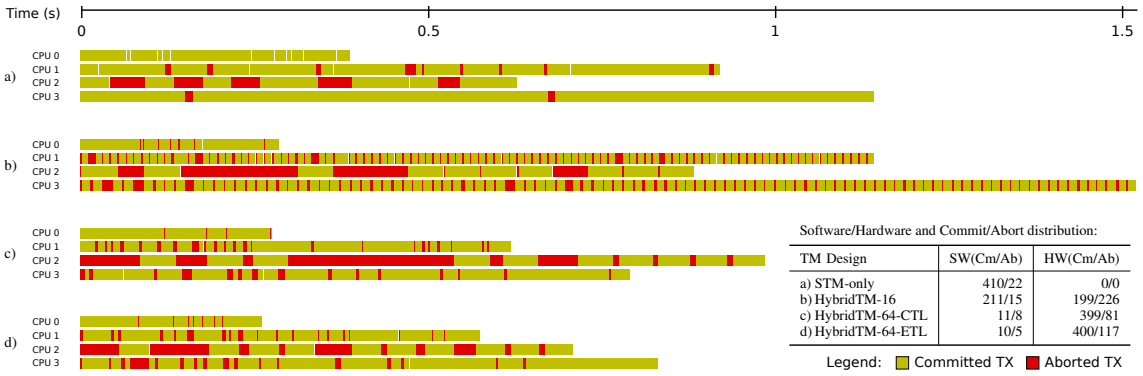


Figure 6.10: Optimizing the execution times of the `intruder` benchmark (execution through Eigenbench) on TMbox. The optimization starts from an STM-only version in Figure 6.10(a), over a hybrid version with 16 entries per TM cache in Figure 6.10(b), through 64 entries per TM cache with commit time locking in Figure 6.10(c), to the hybrid version with encounter time locking in Figure 6.10(d). Figure taken from [189].

In the following, we demonstrate how the visualization helps to refine the execution on the TMbox system [189] and make the assumption that a thread k runs only on CPU k.

Figure 6.10 illustrates the optimization of an hybrid TM application at the example of the `intruder` benchmark mimicked through the corresponding Eigenbench parameter set.

Figure 6.10(a) shows the execution time and behavior with the STM-only version. The gathered statistics reveal many short transactions that fit for a HTM acceleration and long running transactions that require software execution.

Figure 6.10(b) shows the execution of `intruder` with hybrid TM with hardware and software execution of transactions. The execution is longer than with STM-only. The TMbox system supports 16 entries in the TM cache. This is not enough to speedup thread 3 that shows the longest run time. The size of the transactions suggests that increasing the number of entries in the TM cache yields performance improvements.

With 64 entries in the TM cache the execution time is faster than with STM-only (see Figure 6.10(c)). Now thread 3 and thread 1 have a higher amount of transactions executed in hardware. Thread 2 suffers from long, aborted transactions running in software that goes along with a high amount of wasted work. The commit time locking strategy of the STM detects conflicts late so that switching to encounter time locking reduces the amount of wasted work.

Figure 6.10(d) holds the execution with 64 entries in the TM cache and encounter time locking. The execution time from the STM-only version to the final hybrid version has been decreased by 24.1 %. This demonstrates that the visualization and the statistics of the run time behavior of a TM application can be useful to tune the execution of a hybrid TM system.

```
1  #pragma omp for private(i) schedule(static)
2  for (i=0; i<n; i++) { /* sparse q[i] = A*p[i] */ }
3  ...
4  #pragma omp for reduction(+:pq) schedule(static)
5  for (i=0; i<n; i++) { pq += p[i]*q[i]; }
6  ...
7  #pragma omp for schedule(static)
8  for (i=0; i<n; i++) { u[i] = u[i] + alpha*p[i];
9                        r[i] = r[i] - alpha*q[i]; }
10 ...
11 #pragma omp for reduction(+:rr) schedule(static)
12 for (i=0; i<n; i++) { rr += r[i]*r[i]; }
13 ...
14 #pragma omp for schedule(static)
15 for (i=0; i<n; i++) { p[i] = r[i] + beta*p[i]; }
```

Listing 6.11: Mapping of CG to OpenMP for parallelization (extended version of [92, 106, 105]).

6.3 Conjugate Gradients Solver

The algorithm of Conjugate Gradients (CG) is applicable to positive definite matrices to solve the problem $\mathrm{Au} = \mathrm{b}$. Based on the Arnoldi method, the residuals are conjugated to construct the search directions. Therefore search vectors from previous steps must not be stored. For further information regarding the CG method, we refer the reader to the book "Iterative Methods for Sparse Linear Systems" authored by Saad [167]. CG is a very well reseached algorithm that may deliver superlinear convergence [43]. These properties combined with a simple formulation make the method of Conjugate Gradients popular in many research fields. Among these are computational fluid dynamics and the analysis of structural mechanics. These fields employ finite element, finite difference and volume methods for simulation by solving linear systems. In many cases a preconditioner transforms the problem and CG is applied after that. Algorithm 6.1: basic CG algorithm without preconditioning as formulated in [197].

$$\begin{aligned}
\mathbf{q}_k &= \mathrm{A}\mathbf{p}_k, && (6.1)\\
\alpha_k &= \frac{\rho_k}{\mathbf{p}_k \cdot \mathbf{q}_k}, && (6.2)\\
\mathbf{u}_{k+1} &= \mathbf{u}_k + \alpha_k \mathbf{p}_k, && (6.3)\\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k \mathbf{q}_k, && (6.4)\\
\rho_{k+1} &= \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}, && (6.5)\\
\beta_{k+1} &= \frac{\rho_{k+1}}{\rho_k}, && (6.6)\\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k. && (6.7)
\end{aligned}$$

The *scalar product* (also known as *dot product*) is denoted as $\cdot$. Since the vectors exist in an Euclidean space, it can also be called *inner product*. In order to implement the CG

algorithm from the above equations, we need to map it to OpenMP and identify the suited places to employ TM. The *inner product* is the only pattern that requires updating a variable that is shared among threads. Then a single thread performs all scalar computations (as in Equation 6.6). As a consequence only the two reductions qualify for employing TM or other synchronization mechanisms. For dense matrices, the time for the multiplication of the matrix with the vector dominates the overall time for the algorithm. This is different for the sparse matrices that are common in computational fluid dynamics and structural mechanics, in case the number of non-zeros falls in the same complexity class defined through ($nnz \in (n)$) the matrix-vector product as well as the inner product have the same complexity. This increases the relevance of computing the inner product with respect to the overall algorithm and additionally motivates to research the impact of using TM.

The two reductions in normal CG are each implemented in two ways: *Fast* and *Slow*. Fast uses a thread-local variable (e.g., named `pq_priv` in Listing 6.12) to accumulate the results over a private part of the vector that is assigned to this specific thread. Then, a single update adds the thread-local variable to the shared memory one (e.g., named `pq`) that is guarded by a critical section or transaction. Thus, contention between threads only arises from the update of the shared memory variable. In the following we will analyze both patterns for one iteration of the outer loop. The *Fast* version of CG updates one shared memory location per thread and reduction pattern. Thus, the number of executed transactions txn equals the number of threads (th) times the number of reductions red where red equals 2 for normal CG. Hence $txn \sim th$. The *Slow* version updates the shared memory location in one transaction or critical section and does not use a thread-local variable, resulting in one update per iteration of the parallel for loop which is defined through the dimension (dim). Hence $txn \sim dim$. In our case $dim \gg th$ which emphasizes the difference between *Fast* and *Slow* patterns.

Due to the fact that each reduction only updates one shared memory location, OpenMP atomic is a perfect fit for the implementation because it maps to a processor instruction that assures the atomicity of the update (if the processor supports atomics). The *Fast* version, again, uses thread-local variables whereas the *Slow* version does not. This atomicity is limited to one memory location and can not be extended. Thus, the *Atomic Fast* uses the thread-local variables to update the shared memory locations and the *Atomic Slow* updates the shared memory location for each new value. Since each value must be updated with a separate atomic instruction there is no need to distinguish between long and short sections for atomic. These self-made reductions are complemented by the OpenMP reduction, denoted as *Reduction*, that the programmer specifies through using a `#pragma omp for reduction(+:var) schedule(static)`.

```
1 for (i = thread->start; i < thread->end; i++) {
2   pq += p[i]*q[i];
3 }
4 START(thread->id, RW);
5   pq_priv += (double)LOAD_DOUBLE(&pq);
6   STORE_DOUBLE(&pq, pq_priv);
7 COMMIT;
```

Listing 6.12: *Fast* version of a reduction that is implemented with TM macros (similar to [92, 106, 105]).

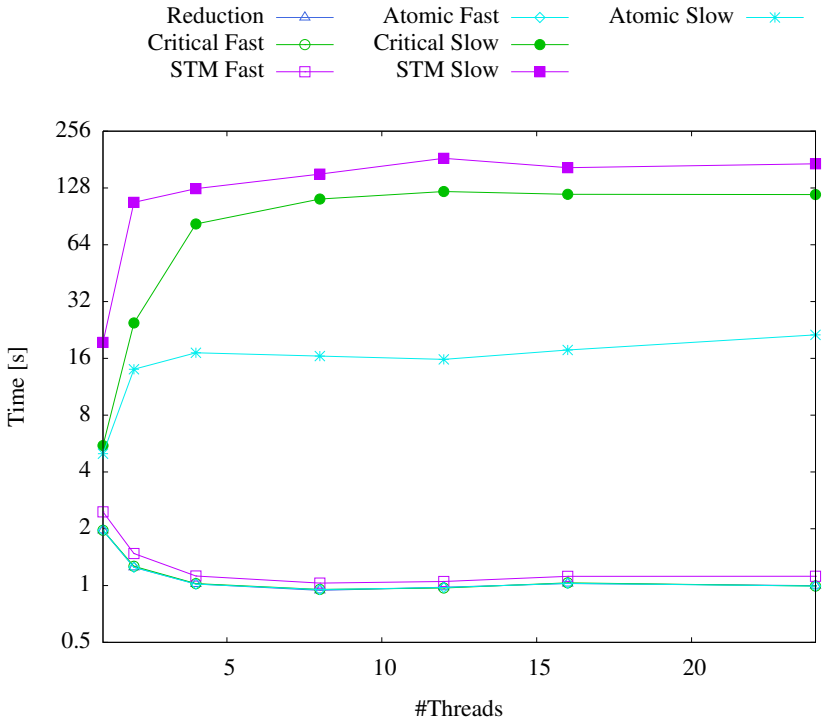


Figure 6.11: Execution time of normal CG with 1 to 24 threads on ExpX5670.

Figure 6.11 shows the run time of the normal CG on the y-axis over the number of threads on the x-axis. Please note that the run times of the *Reduction*, *Critical Fast*, and *Atomic Fast* patterns are nearly identical so that only *Atomic Fast* is readable. These cases show the lowest execution time as the number of threads increases. The run times of *STM Fast* follow the same trends but with a small offset that indicates a longer execution time. All *Fast* variants significantly outperform the *Slow* variants: *Atomic Slow* performs best among them with an moderate increase in execution time for higher thread counts. *Critical Slow* and *STM Slow* perform worse with *STM Slow* having the longest execution time with 8 threads. None of the *Slow* variants reaches the respective single thread execution time.

In an earlier version of this work, we found a peak in the run time for 24 threads with OpenMP [92, 106, 105]. The dimension of the underlying matrix is set to $5M$ in this case. The execution time did not differ much with eight threads on ExpX5670. A special case is 24 threads and OpenMP: the calculation takes slightly longer than with a single thread. A model that describes the effects of the scheduling on the run time of the application explains the peak with 24 threads. Christmann et al. developed this model for researching the impact of oversubscription on the application throughput [37]. The scheduling algorithm must fulfill the following three criteria:

1. the scheduling must be fair such that each process gets a fair share of time,
2. it has to balance the load across cores (or hardware threads),
3. and it must pin a process to a processing element as long as possible.

For a system with N processing elements that already runs r threads, their formula estimates the minimum δ_{min} and maximum δ_{max} occupation of a processing element when a second application runs x threads. Assuming that the three scheduling criteria are satisfied, the following formulas describes the occupation of a single processing

element [37]: $\delta_{min} = max(\lfloor \frac{r+x}{N} \rfloor, 1)$ and $\delta_{max} = \lceil \frac{r+x}{N} \rceil$. The actual occupation δ ranges in between δ_{min} and δ_{max}. The following formula estimates the number of processing elements that will be granted to an application with x threads: $n(x, \delta) = min(\frac{x}{\delta}, N)$. which is limited by N processing elements. In the paper, Christmann et al. further apply a linear transformation and show that the model matches their experiments. In the following, we argue that this model also explains the scheduling artifacts that we observed in previous experiments [92, 106, 105]. Our previous case with the Linux kernel 2.6.32-29-server meets all of these assumptions. The explanation for the peak in execution time is that a fully loaded node (with 24 OpenMP threads) competes with some background process for computing resources. Eventually, after a long stall time, one of the OpenMP threads gets migrated to a different processing element. The combination of these tasks, including the long stall until the scheduling decision is made, leads to a prolonged overall execution time. The experiments presented here, verify that a later Linux kernel (with version number 3.0.0-23-server) that enables a fair scheduling of groups instead of processes does not show this behavior anymore.

6.3.1 Pipelined Conjugate Gradients Solver

Inspired by Meurant's algorithm [135], Strzodka and Göddeke refine the pipelined Conjugate Gradients solver to enable mixed precision and pipelined algorithms that accurately solve partial differential equations with low precision components on FPGAs [197]. From these collection of proposed algorithmic variants of the conjugate gradient method, we choose the basic pipelined CG variant because some of its properties seem worthwhile to explore in the context of synchronization with transactional memory. The idea behind the pipelined CG is that all computations on vector elements should be done in parallel. With these rearrangement it becomes feasible to stream a vector instead of having to store all elements of the vector. First, Strzodka and Göddeke reorder all vector operations so that these can be performed in parallel [197]:

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{p}_k, \tag{6.8}$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{q}_k, \tag{6.9}$$

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k, \tag{6.10}$$

$$\mathbf{q}_{k+1} = \mathrm{A}\mathbf{p}_{k+1}. \tag{6.11}$$

The main contribution of this algorithm is to eliminate the requirement to compute all elements of the vector $\mathbf{r}_{k+1}$ in order to compute $\mathbf{p}_{k+1}$. Strzodka and Göddeke lift this restriction through introducing $\sigma_k = \rho_{k+1}$ that does not require knowledge of $\mathbf{r}_{k+1}$: $\sigma_k = \alpha_k(\alpha_k \mathbf{q}_k \cdot \mathbf{q}_k - \mathbf{p}_k \cdot \mathbf{q}_k)$. Then the scalar products are computed after the reordered vector operations shown in Equation 6.8:

$$\rho_{k+1} = \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}, \tag{6.12}$$

$$\alpha_{k+1} = \frac{\rho_{k+1}}{\mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}}, \tag{6.13}$$

$$\sigma_{k+1} = \alpha_{k+1}(\alpha_{k+1} \mathbf{q}_{k+1} \cdot \mathbf{q}_{k+1} - \mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}), \tag{6.14}$$

$$\beta_{k+1} = \frac{\sigma_{k+1}}{\rho_{k+1}}. \tag{6.15}$$

This variant is useful in the case of a sparse matrix A that enables to compute one step of the algorithm in a fully pipelined fashion. The pipelined CG variant is suited in case the

```
1  do {
2  #pragma omp parallel
3    {
4  #pragma omp for private(i) schedule(static)
5      for(i=0; i<dim; i++) u[i] += alpha*p[i];
6  #pragma omp for private(i) schedule(static)
7      for(i=0; i<dim; i++) r[i] -= alpha*q[i];
8  #pragma omp for private(i) schedule(static)
9      for(i=0; i<dim; i++) {
10       p[i] = r[i] + beta*p[i]; q[i] = 0;
11     }
12 #pragma omp for private(i,j,start,end,temp)\
13                 schedule(static)
14     for(i=0; i<dim; i++) {
15       start = row[i]; end = row[i+1];
16       for(j=start; j<end; j++) {
17         temp = col[j]; q[i] += val[j]*p[temp];
18       }
19     }
20 #pragma omp single
21     {
22       rho = 0.0; pq = 0.0; qq = 0.0;
23     }
24 #pragma omp for reduction(+:rho)\
25                 reduction(+:qq)\
26                 reduction(+:pq) schedule(static)
27     for(i=0; i < dim; i++) {
28       rho += r[i] * r[i];
29       qq  += q[i] * q[i];
30       pq  += p[i] * q[i];
31     }
32 #pragma omp single
33     {
34       alpha = rho/pq;
35       sigma = alpha * (alpha*qq - pq);
36       beta  = sigma / rho;
37     }
38   } /* End of OpenMP parallel region. */
39   iteration++;
40   norm_r = sqrt(rho);
41 } while (norm_r > epsilon);
```

Listing 6.13: Implementation of the pipelined CG with OpenMP Reduction.

matrix A is sparse and does not require global communication. Therefore, the pipelined CG is e.g., applicable for solving the stationary heat equation without heat source.

Listing 6.13 shows the main loop of the implementation of the pipelined CG algorithm with OpenMP. This loop, starting from line 1, iterates until $|\mathbf{r_{k+1}}| <= \epsilon$ being the convergence criteria for the algorithm. The algorithm converged to a solution when it found a solution that satisfies this condition. In the following, we will discuss the mapping of the algorithm from the Equation 6.8 and Equation 6.12 to this implementation. First, $\mathbf{u}_{k+1}$ and $\mathbf{r}_{k+1}$ are both computed according to Equation 6.8 and Equation 6.9 in line 4 and 5 ($\mathbf{u}$) and 6 and 7 ($\mathbf{r}$) in an OpenMP for loop. Then line 8 to 11 perform the computation of p_{k+1} as described in Equation 6.10 and reset the vector $\mathbf{q}$. The sparse matrix multiplication, involving A and $\mathbf{p}$, takes place in the lines 12 to 18 according to Equation 6.11. The implementation resets the scalar variables in line 19 to 22. Then, the three reductions compute the inner products $\mathbf{r_{k+1}} \cdot \mathbf{r_{k+1}}$, $\mathbf{p_{k+1}} \cdot \mathbf{q_{k+1}}$, $\mathbf{q_{k+1}} \cdot \mathbf{q_{k+1}}$ in lines 23 to 30. In comparison with the CG according to Saad [167], that demanded two separate reductions, the computation with pipelined CG requires three reductions. The advantage is that one enlarged critical section or transaction embraces all three of them. These three reductions implement the vector operations of Equation 6.12, Equation 6.13, and Equation 6.14. Please note that all of the above steps except resetting the scalar variables are performed in parallel. Computing Equation 6.15 again requires to serialize the execution (using `#pragma omp single` statement in line 31) and compute the values of $\alpha_{k+1}, \sigma_{k+1}$ and β_{k+1}. Another way of avoiding this serialization would have been to end the parallel section in line 31 instead of line 37. However, this minor detail does not make a large difference. Finally line 38 (already outside the parallel section) increments the number of iterations and line 39 computes the norm of r_{k+1}. The result is compared with ϵ in the while statement. In case the norm of r_{k+1} already satisfies the condition, the implementation will output the result and also compute the error (not shown in Listing 6.13. Otherwise, the algorithm performs another iteration of the loop until the solution satisfies the condition.

The implementation in Listing 6.13 reveals that our approach does not take advantage of the fact that pipelined CG supports the streaming of vectors. Instead our approach aims to implement the reduction, that can now be made three times larger than before, assuming a constant vector size. This different reduction pattern enables us to use larger transactions (or critical sections) and again to implement them in two different ways. These different ways of implementing the reduction pattern are compared and analyzed in the following. The *Reduction* case uses the OpenMP reduction concatenating three reductions in one pragma: `#pragma omp for reduction(+:rho) reduction(+:qq) reduction(+:pq) schedule(static)`. The three reductions are implemented in three ways: *Fast*, *Slow Long* and *Slow Short*. *Fast* executes the accumulation with a thread-local variable over a thread private part of the vector. After finishing this calculation, one update adds the thread-local variable (e.g., `pq_priv`) to the shared memory one (e.g., `pq`). Thus, contention between threads stems from the update of the shared memory variable. Regardless of the dimension this pattern leads to th updates of the shared variable which is a huge gain compared with the *Slow* pattern. Of course the complexity to implement the *Fast* pattern is slightly higher. We expect this variant to perform better and therefore label it *Fast*. The *Fast* version of pipelined CG updates all three shared memory variables in one transaction/critical section. This enlarges the size of the transaction because instead of one update with normal CG for each of the reductions, there are three updates in pipelined CG. *Slow Long* updates the three shared memory locations in one transaction or critical section and does not use a thread-local variable for storing intermediate results. *Slow Short* also

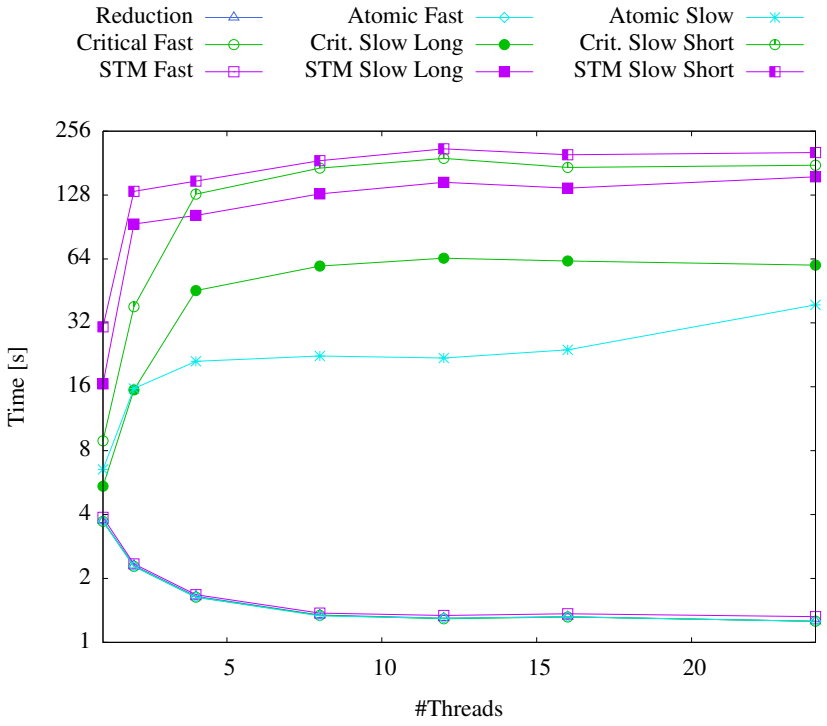


Figure 6.12: Execution time of pipelined CG with 1 to 24 threads on ExpX5670. Figure as in [176].

does not use thread-local variables to store intermediate results and performs each update of a shared memory location in a dedicated transaction or critical section. Thus, both *Slow* variants require the same amount of updates of the shared variable and differ only in the granularity of the applied synchronization mechanism. Assuming a dimension of dim, the shared variable is updated dim times. If the work is distributed evenly among th threads, each threads performs $\frac{dim}{th}$ updates. For $dim >> th$ this pattern creates contention on the shared variable because each thread needs to access it multiple times. In a multi-core system this will result in coherency traffic that will invalidate the datum in the other caches, leading to performance loss. For TM this leads to an increasing number of rollbacks. Since we expect this variant not to perform as good as the *Fast* variant, we denote it as *Slow*. For OpenMP atomic the *Fast* version uses thread-local variables whereas the *Slow* version does not. If possible `omp atomic` maps to a native atomic instruction that updates one memory location without being interrupted by other processors. This atomicity is limited to one memory location and can not be extended. Thus, the *Atomic Fast* uses the thread-local variables to update the shared memory locations and the *Atomic Slow* updates the shared memory location for each new value. Since each value must be updated with a separate atomic instruction there is no need to distinguish between long and short sections.

The execution time of the pipelined CG method is depicted in Figure 6.12. The y-axis holds the average execution time in seconds over 17 runs and the x-axis the number of threads. Again, the *Fast* versions of *Atomic*, *Critical* and *Reduction* show a nearly identical behavior and are hard to distinguish. *Fast STM* also has a slightly higher execution time than e.g., *Reduction* but the gap appears smaller than before. The ranking of the slow variants is as follows: *Atomic Slow Long*, *Critical Slow Long*, *STM Slow Long*, *Critical Slow Short*, *STM Slow Short*. Again, neither slow variant achieves the run time of respective single thread. Thus, all slow variants show a slowdown for the execution with more than two threads.

Dimension	Epsilon	Start vector	Solution
$5 * 10^6$	$1 * 10^{-13}$	$\vec{0}$	$\vec{1}$

Table 6.2: Parameter settings for the example problem solved with the two implementations of the conjugate gradients method. Table similar to [176].

The interesting insight is that neither of the short variants (STM or critical) performed as good as or better than a long variant. Thus, enlarging the granularity of critical sections under the given conditions results in a better performance but the only way to achieve a speedup is the use of thread-local variables that avoid frequent updates of the shared memory and hereby reduce contention for shared locations.

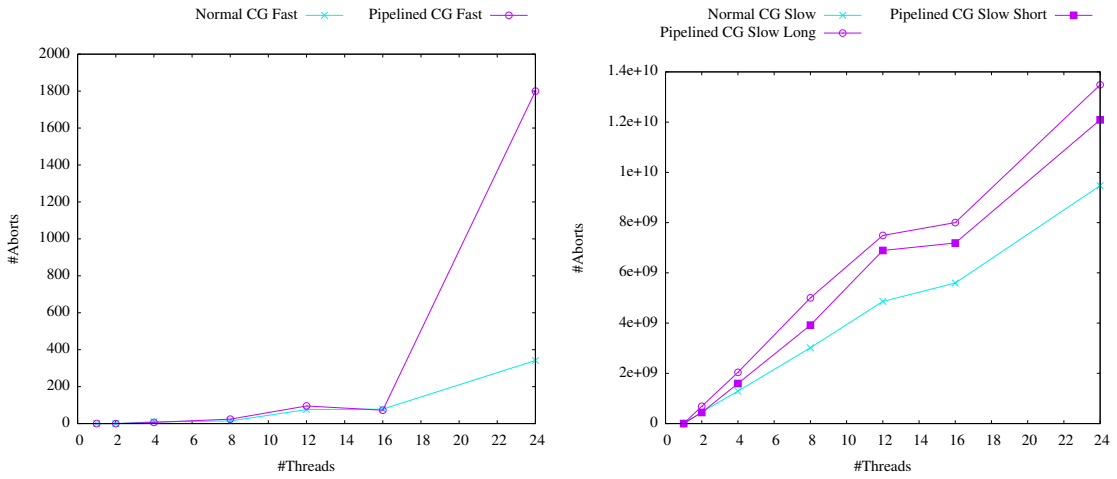
6.3.2 Comparison of CG and Pipelined CG

In order to compare normal CG and pipelined CG, that are both capable of solving systems of linear equations with symmetric and positive-definite matrices, we apply them to an example that describes the steady state conduction of heat with the method of finite differences. This example solves the stationary heat equation without heat source in one dimension. The key parameters for the following experiments are shown in Table 6.2. Both CG variants execute a loop that iterates over the numerical algorithm. Each iteration refines the current solution and herewith reduces the error. The error is computed as the euclidean norm of $\mathbf{b} - \mathrm{A}\mathbf{u}_k$. The algorithm iterates as long as the error is greater than epsilon. In case a solution exists that satisfies the criteria, the process is also said to *converge* to this solution. In practice the convergence may be perturbed through round-off errors that affect the numerical stability. Whether an algorithm is suited to find a solution to a given problem also depends on the algorithmic details as well as the implementation. Thus, a different formulation of the same algorithm may show a different convergence behavior. Moreover, even implementation details such as the order of elements when summing up a vector, may have an impact on the convergence behavior. Thus, the impact of new technologies, like TM, on the convergence behavior has to be researched thoroughly. We implement both CG variants in the programming language `C` and parallelize them, as described earlier, with OpenMP and the described synchronization mechanisms. GCC in version 4.6.1 generates both executables with the compiler options `-fopenmp -O3 -g3` that affect performance.

Software Transactional Memory Characteristics

The transactional characteristics of both CG variants are discussed first because these may dominate the utilization of architectural resources. For example a transaction that performs mainly integer operations and aborts frequently and, thus, repeats these operations multiple times contributes a larger share of integer operations than a transaction that successfully commits. Hence, large abort rates may change the heavily utilized functional units. Figure 6.13 depicts the absolute number of aborted transactions of all threads for the *Fast* (left hand side) and the *Slow* variants (right hand side) of normal CG and pipelined CG. All of the presented numbers are averages over 17 runs.

For the *Fast* version (illustrated in Figure 6.13(a)) the number of aborts is below 100 for up to 16 threads and normal CG and pipelined CG. With 24 threads it rises to ≈ 340 for normal CG and ≈ 1800 for pipelined CG. This is the only configuration for the *Fast* versions that normal CG has significantly less aborts than pipelined CG. This is remarkable because



(a) Aborts with *Fast* versions of normal and pipelined CG

(b) Aborts in *Slow* versions of normal and pipelined CG

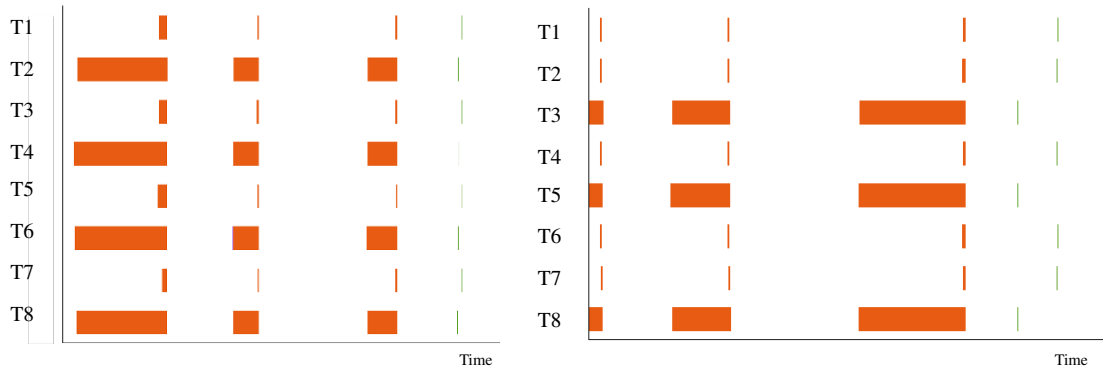
Figure 6.13: Aborts with normal and pipelined CG with the number of threads ranging from 1 to 24 on ExpX5670. Figure similar to [176].

pipelined CG executes three times the number of loads and stores per transaction and herewith should have a higher probability of conflict. The fact that all of these transactions access the same three variables in the same order lead to a scenario where a transaction will conflict with another transaction if they both run at the exact same time. Because pipelined CG requires more time to execute the longer transactions, this increases the conflict probability because a longer time inside a transaction although means a longer time in which a second thread may start a transaction and conflict with the former. This effect is dominating only at 24 threads because prior to that, both versions of CG perform equal. The relative abort rate for *Fast* with 8 threads is $\approx 3.5\,\%$ for normal CG and $\approx 6\,\%$ for pipelined CG.

Figure 6.13(b) demonstrates the reason for the missing performance with the *Slow* variants. For only 2 threads normal CG already has $\approx 460 * 10^6$ aborts. For pipelined CG the aborts for 2 threads are $\approx 440 * 10^6$ for the short and $\approx 687 * 10^6$ for the long variant. The reason for these high aborts are the transactions that update a single variable (or in the best case three variables) for each iteration of the loop. Due to these high abort numbers, the threads will not make progress and, hence have long execution times and poor performance with the *Slow* variants.

After studying the transactional characteristics in the previous paragraph, we would like to demonstrate the additional values and the flexibility of the VisOTMA framework by doing an in-depth analysis of the *Fast* versions of normal CG and pipelined CG. When first visualizing the TM application behavior with Paraver, we found many gaps between extremely small transactions. Thus, only a high zoom level would allow us to find the aborted transactions. After investigating these cases and finding that the overall TM performance for *Fast* is good (also cf. to previous paragraph), we decided to focus on the blank spots between the transactions. A code study reveals that, apart from the computational parts, OpenMP constructs are most likely to consume the missing time in between the transactions.

Listing 6.11 and Listing 6.13 show that both CG variants perform 5 OpenMP `for` loops. By default these `for` loops come with an implicit barrier at the end of the execution. Thus,



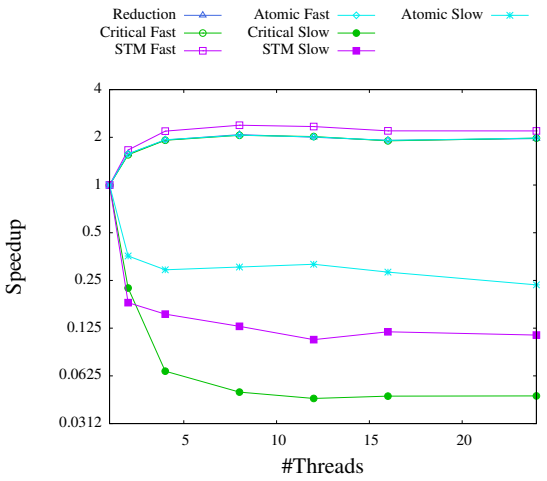
(a) Visualization of normal CG

(b) Visualization of pipelined CG

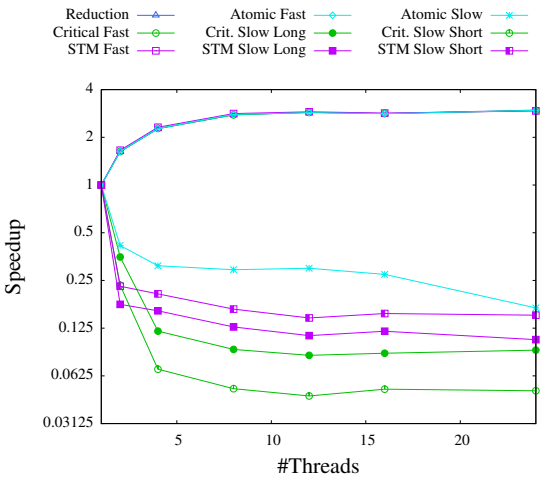
Figure 6.14: Visualization of normal and pipelined CG with 8 threads on ExpX5670. Zoomed to relate transaction time (in green) with barrier wait time (in orange). Figure similar to [176].

the fastest thread waits for the slowest one to complete its work and reach the barrier. OpenMP enables the programmer to specify the `nowait` clause to omit this barrier [44]. On the other hand there is also the explicit `#pragma omp barrier` construct that produces a barrier. These manifold possibilities to generate barriers in OpenMP code and the importance for CG code, convinced us to investigate the barrier wait times to relate these to the transaction execution times. In our particular setup using the GCC compiler, `libgomp` is the OpenMP run time system and GCC does the expansion of OpenMP pragmas and the outlining of functions. In order to implement timed barriers, we needed to intercept the call to the original barrier call with one that would record the cycle counter of the processor on entry and exit of the barrier. These readings are then written to a thread-local trace file. Using the timed instead of the regular barriers is achieved through a simple replacement on assembly level. Through simply replacing the call to *GOMP_barrier* with a call to *ote_GOMP_barrier*, we achieved the desired functionality. Thus, *ote_GOMP_barrier* records the cycle counter before and after calling *GOMP_barrier*. Separate additional trace files for tracing these barriers are necessary because barriers do not depend on transactions and the STM may not be initialized when calling a barrier. Thus, a post-processing step merges the barrier traces with the TM traces. Both trace files have the same time base and, hence, timed correlation and the visualization of the merged traces requires to register the new events at the various processing stages but is straightforward.

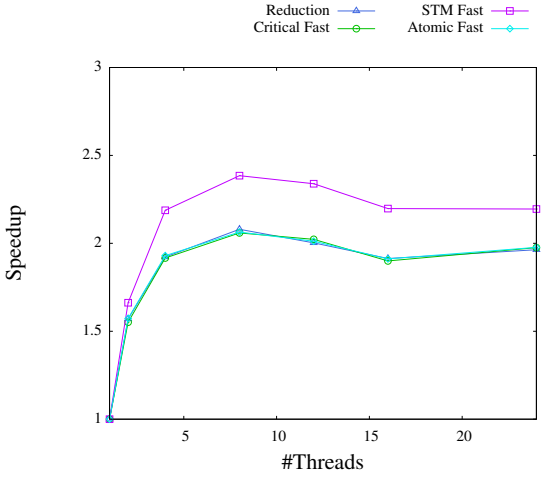
Figure 6.14 shows results of this effort for normal CG and pipelined. The picture holds a timeline view of barriers and transactions executed by 8 threads. The threads, denoted with `T1` to `T8` each occupy a slot on the y-axis. The x-axis shows the progress of time. The orange bars demonstrate the wait time of a particular thread at a barrier. These orange bars dominate the Figure 6.14(a). Green bars illustrate how much time the execution of a transaction with a commit takes. These bars are present only on the right hand side of Figure 6.14(a) and are extremely small. Figure 6.14(b) illustrates the run time behavior of the pipelined CG variant that exercises a similar execution pattern. Again transaction times are hardly visible although these transactions have three times the amount of loads and stores of those transactions in normal CG.



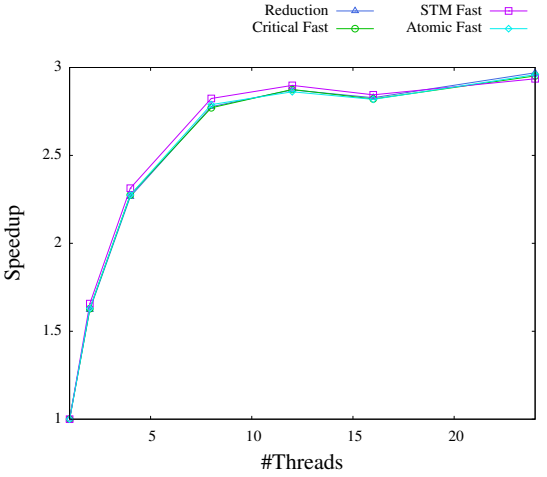
(a) Speedup of normal CG



(b) Speedup of pipelined CG



(c) Speedup of normal CG for fast versions only.



(d) Speedup of pipelined CG for fast versions only.

Figure 6.15: Speedup with normal and pipelined CG with the number of threads ranging from 1 to 24 on ExpX5670. Figure similar to [176].

Additionally, we discover that pipelined CG shows a small perturbation that influences the start time of the transactions. Whereas in Figure 6.14(a) with normal CG all threads start their transaction at almost exactly the same point in time, Figure 6.14(b) reveals that three threads start executing the transaction before all other threads. This behavior of pipelined CG is likely the cause for a better conflict rate than expected. Surprisingly both figures highlight that the wait times at the barriers (colored in orange) exceeds the execution time of a transaction (shown in green). These findings not only motivate research to avoid or omit these barriers but also show that in a very regular setting, such as with the CG algorithm, TM can not show its strong side because the effects of optimistic concurrency, that enable some threads to proceed faster than others are potentially turned into wait times at the barriers, waiting for the slowest thread that has been aborted to enable the progress of the fast threads.

Speedup

This paragraph compares the achieved speedup of normal CG with that of pipelined CG. The speedup is computed according to $S(n) = \frac{T(1)}{T(n)}$, where $T(n)$ denotes the execution

time with n threads and $T(1)$ is the respective single thread execution time (cf. to [86]). Often T_{seq} is used instead of $T(1)$ with T_{seq} being the serial reference implementation that does not incur the overheads of a threaded implementation. In these cases often $T_{seq} < T(1)$ holds so that the resulting $S(n)$ would be smaller.

Figure 6.15(a) depicts the speedup for normal CG whereas Figure 6.15(b) shows it for pipelined CG (both times on the y-axis). The x-axis holds the number of threads. Although the plots of the runtimes from previous sections contain the same information, this plot more evidently shows a slow down (speedup < 1) for the slow variants and a speedup for the fast variants. Setting the scale of the y-axis is a compromise to fit all variants on one plot. This makes identifying the maximum achieved speedup difficult because of the low resolution in this segment. Therefore, a second plot focuses on displaying the results of the fast variants only.

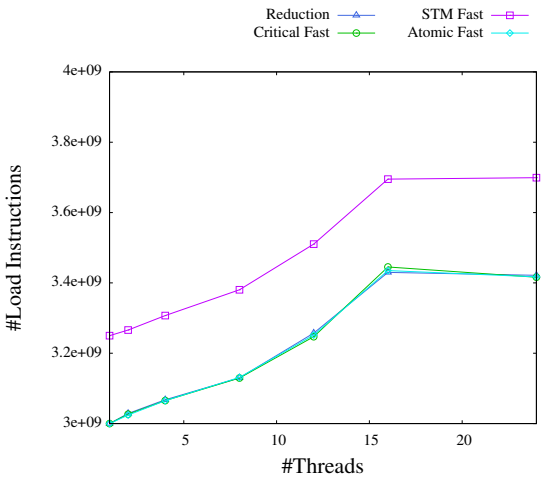
Figure 6.15(c) and Figure 6.15(d) show the speedup with a linear scale on the y-axis. This plot illustrates that the achievable speedup over the respective single thread performance is higher with pipelined CG, achieving the highest speedup of 2.97 with the regular reduction and 24 threads. For normal CG, *STM Fast* with 8 threads achieves the best speedup of 2.38. The overhead of the single thread execution has a large influence on the reported speedups because a larger overhead (e.g., with STM) leads to a slower execution time. If the speedup is computed relative to this single thread execution time, this yields a higher speedup because the baseline has been worse. This effect could be avoided by having a fixed serial execution time for all benchmarked variants. Here, this effect leads to the situation that *STM Fast* has a higher speedup for e.g., pipelined CG with 8 threads but a higher execution time than e.g., *Reduction*.

Convergence Behavior

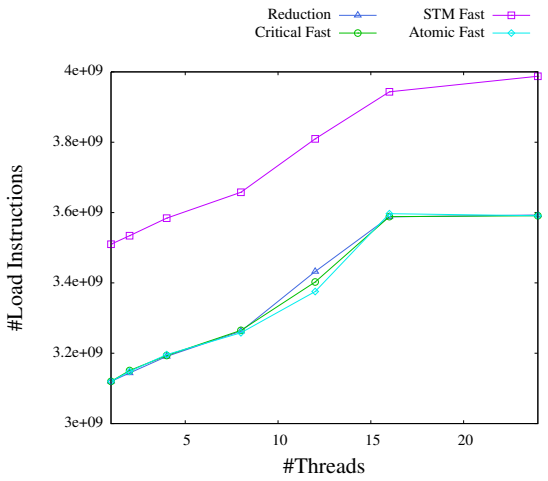
This paragraph presents the results of examining the convergence behavior of normal CG and pipelined CG applied to a problem that solves the stationary heat equation without heat source. The parameters setting is identical with the one that has been shown in Table 6.2. Both variants of CG show a consistent convergence behavior across all tested thread counts and synchronization mechanisms. Normal CG converges after 25 iterations to a solution that satisfies the criteria. Pipelined CG finds a solution to the problem that satisfies the convergence criteria after 26 iterations. Therefore, even for the simple problem chosen in this experiment, the choice of the implementation strategy has a huge impact on the convergence behavior. Pipelined CG needs to perform one additional iteration which is equal to a relative increase of computational complexity of $4\,\%$.

Utilization of Architectural Resources

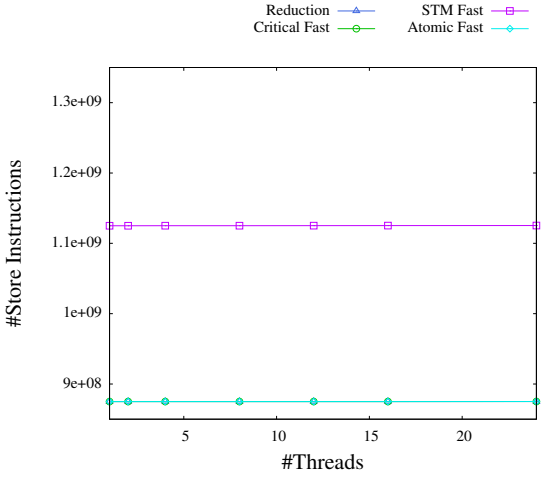
In the following, we will inspect the utilization of architectural resources by the synchronization mechanisms as well as the two different algorithms. Pipelined CG excercises a different compute pattern and has to perform one additional iteration to find a solution that satisfies the convergence criteria for the selected problem. These changes are going to impact the number of retired instructions, the number of loads and stores, the number of floating point operations, the branch prediction and the use of the synchronization primitives. We will highlight the differences in this section and research the synchronization mechanisms for each of the algorithms.



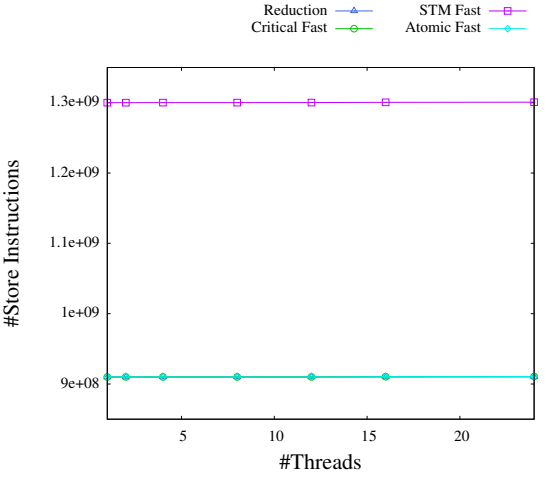
(a) Normal CG load instructions



(b) Pipelined CG load instructions



(c) Normal CG: store instructions executed

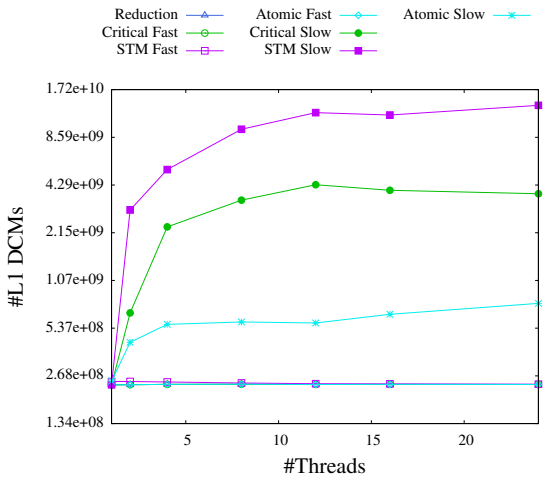


(d) Pipelined CG: store instructions executed

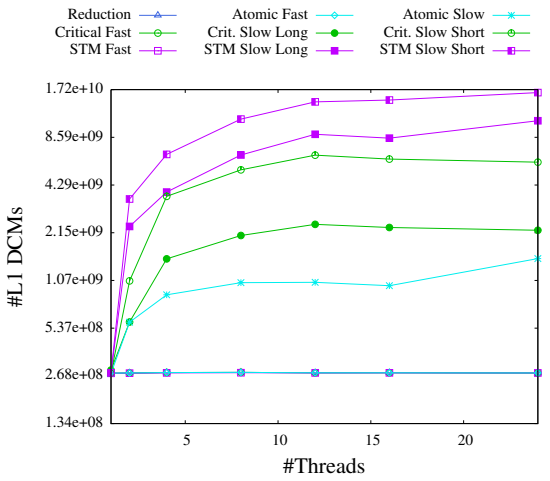
Figure 6.16: Load and store instructions with both CG variants on ExpX5670.

Influence of thread count on executed loads and stores. Figure 6.16(a) depicts the number of load instructions on the y-axis and the number of threads on the x-axis for normal CG. The number of loads increases with the number of threads up to 16 for all synchronization variants. For 24 threads the number of loads is similar to the case with 16 threads. *STM Fast* performs 8 % more load instructions than the other fast variants with 8 threads. Figure 6.16(b) highlights the same trend for pipelined CG. For 8 threads and pipelined CG *STM Fast* executes 12 % more loads instructions than *Reduction*. Pipelined CG shows an average increase in loads over normal CG that ranges between 4 % (for *Reduction*, *Critical Fast* and *Atomic Fast*) and 8 % (for *STM Fast*). This difference can be explained by executing the additional iteration to achieve convergence.

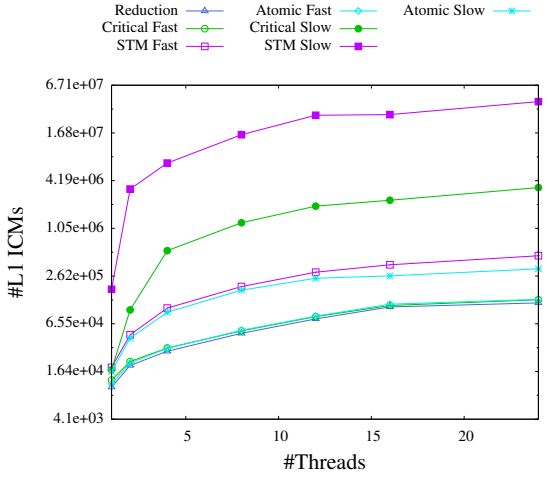
The y-axis of Figure 6.16(c) reveals the number of store instructions of normal CG whereas the x-axis illustrates number of threads. The number of stores is constant for all thread counts and on average 28 % higher for *STM Fast* compared with the other fast variants. Figure 6.16(d) illustrates the same trend for pipelined CG: *STM Fast* carries 43 % more store instructions on average than the other fast variants. On average normal CG executes 4 % more stores than pipelined CG and *Reduction* and 16 % more stores for *STM Fast*. The result for *Reduction* perfectly matches the expected increase due to the additional interation



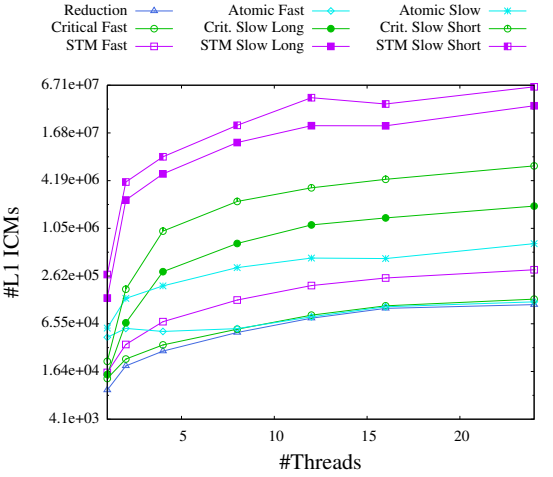
(a) Normal CG and L1 data cache misses



(b) Pipelined CG and L1 data cache misses



(c) Normal CG showing L1 instruction cache misses



(d) Pipelined CG showing L1 instruction cache misses

Figure 6.17: Level 1 instruction and data cache misses for both CG variants on ExpX5670.

that is required to satisfy the convergence criteria. Further, an overall insight is that the number of stores is independent of the number of threads whereas the number of loads increases with the number of threads.

L1 data and instruction cache misses are of major importance for the performance of an application as each load exercises the memory hierarchy to bring the datum to the processor. In case of a L1 cache miss, the datum must be retrieved from the second level cache that has a higher latency and results in a slower execution.

For the L1 data cache, Figure 6.17(a) and Figure 6.17(b) show the total number of L1 data misses on the y-axis over an increasing number of threads on the x-axis for normal CG and pipelined CG respectively. Both show a similar trend for the slow and fast variants of the synchronization mechanisms: the fast variant does not increase significantly with more threads - it stays constant. The slow variants reveal a significant increase that appears to follow a logarithmic function with an increasing number of threads. The *Reduction* pattern with 4 threads shows $2.37 * 10^8$ L1 data cache misses with normal CG whereas pipelined CG generates $2.81 * 10^8$ misses. This is significantly more than a relative increase of

$4\,\%$ that is explained by the additional iteration to reach convergence (cf. Section 6.3.2). Therefore, the algorithmic structure of pipelined CG puts a higher pressure on the L1 data cache.

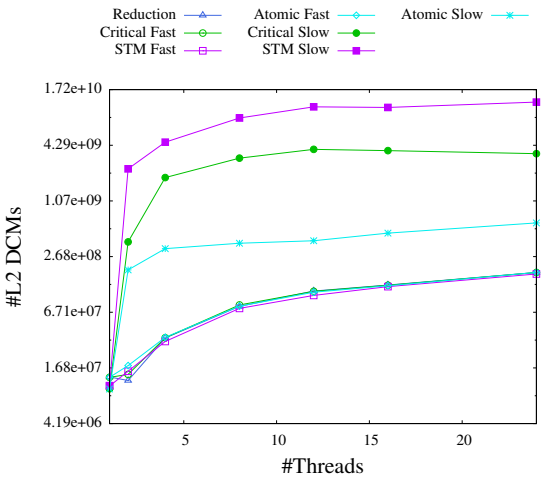
The utilization of the L1 instruction cache is similar to the L1 data cache. Figure 6.17(c) holds the number of L1 instruction cache misses on the y-axis and the number of threads on the x-axis for normal CG. *STM Slow* and *Critical Slow* have the highest counts that seem to follow a logarithmic function. The interesting result is that *Atomic Slow* has a lower instruction miss count than *STM Fast*. This means that the overheads of STM are more prominent on the L1 instruction than on the L1 data cache. *Atomic Fast* and *Reduction* yield the lowest L1 instruction cache misses. Figure 6.17(d) shows a similar plot for pipelined CG. Similarly, *STM Slow Short* and *STM Slow Long* have the highest L1 instruction misses followed by *Critical Slow Short* and *Critical Slow Long*. In this case, *Atomic Slow* does not perform as good as *STM Fast* but the overheads with STM are still significant compared to *Critical Fast* and *Reduction*. Interestingly, *Atomic Fast* shows a peak in the miss rate for two threads that is higher than the misses for *STM Fast*.

The poor performance of the slow variants in L1 instruction as well as data caches, that becomes even worse with an increasing number of threads, explains the observed slowdown for these implementations. This observation holds for both CG variants.

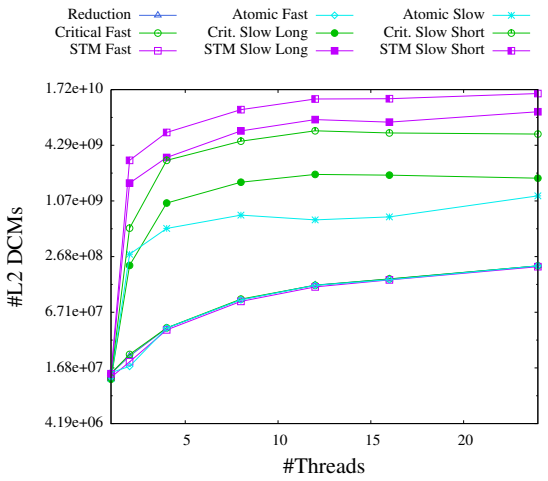
L2 data cache misses are important for the performance because a cache miss needs to be retrieved from the L3 cache. Resulting in a even larger penalty. Similarly, the number of access of the L2 cache equals the number of misses in the L1 data cache and has been studied in the previous paragraph. Therefore, we focus on L2 data cache misses. Figure 6.18(a) holds these for normal CG where Figure 6.18(b) shows L2 data cache misses for pipelined CG. The results for both are similar to the L1 cache misses and can be summarized as follows: the slow variants show an increase that resembles a logarithmic function as the number of threads rises. The fast variants also show an increase in L2 cache misses for higher thread counts but with smaller slope. In order to ease the understanding of these plots, Figure 6.18(c) and Figure 6.18(d) show the respective L2 cache miss rates. The findings are consistent with the previous observations and show that the slow variants show a steep increase (that resembles a logarithmic function) as the thread count rises, whereas the fast versions show a linear increase. For both CG variants the cache miss rate is above $76\,\%$ for 24 threads.

The fact that the L2 cache miss rate increases with higher thread counts leads to longer wait times for data and, thus, limits the scalability of both CG variants. More importantly this behavior is consistent across CG variants as well as synchronization mechanisms and highlights the importance of the latency of these data memory accesses. In the case of synchronizing threads with STM, additional L2 misses may be due to the higher number of aborts at higher thread counts. However, the comparison with other synchronization mechanisms, that show a similar L2 behavior as STM, demonstrates that the higher number of aborts is not the dominating factor.

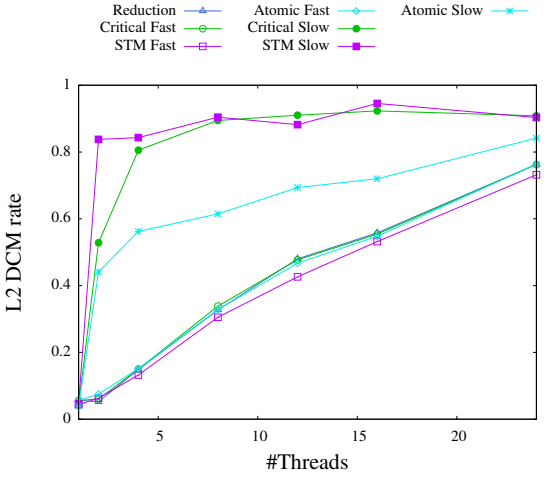
Studying the instructions retired and floating point instructions. Due to the subdue performance of the slow variants, we will focus the following discussion on the fast variants only. Figure 6.19(a) depicts the retired instructions for normal CG. *STM Fast* retires $3.3\,\%$ more instructions than *Critical Fast* with 8 threads. This again, is due to the large overheads associated with executing shared memory accesses through a STM library that performs conflict detection. For pipelined CG, the trend is similar (cf. to



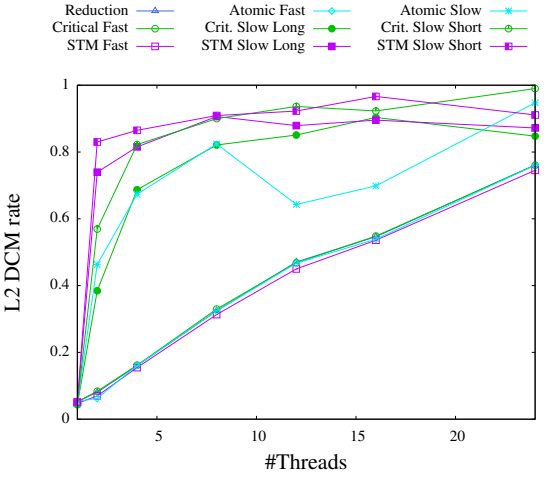
(a) L2 data cache misses of normal CG



(b) L2 data cache misses of pipelined CG



(c) L2 data cache miss rate of normal CG

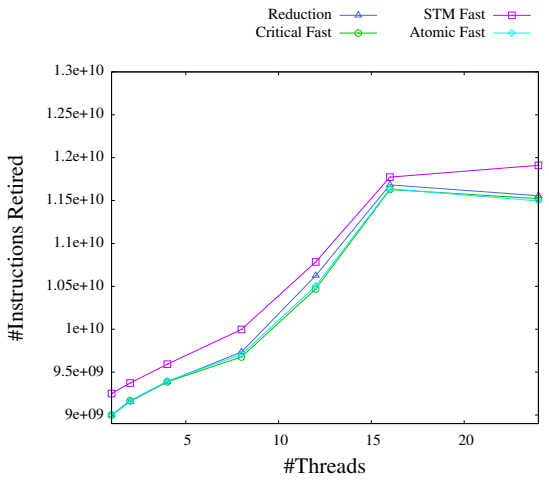


(d) L2 data cache miss rate of pipelined CG

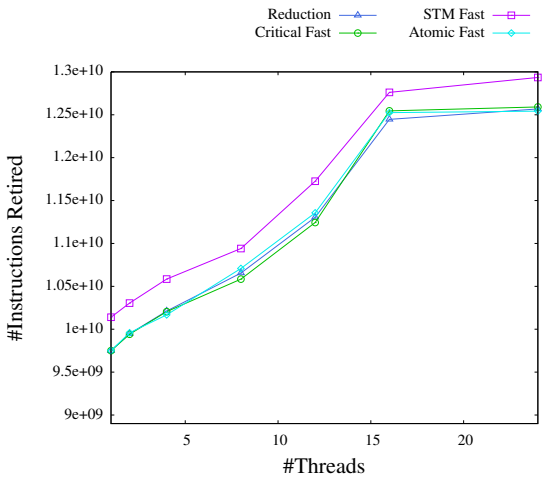
Figure 6.18: Data cache misses in L2 for both CG variants on ExpX5670.

Figure 6.19(b)) but it also becomes apparent that pipelined CG requires more instructions than normal CG. An increasing number of threads retires more instructions (up to 16 threads). The explanation is that each of the threads executes the same functions that have been outlined by the OpenMP passes of the compiler. Thus, each new thread increases the number of instructions retired. Then, for 24 threads the number of retired instructions stays constant (pipelined CG) or decreases (normal CG but not for *STM fast*). To correlate these measurements with the TM statistics is difficult because the higher number of aborts measured with 24 threads does not correlate with a smaller number of instructions retired. One might suspect an aliasing of performance counters that leads to a smaller number of instructions being reported.

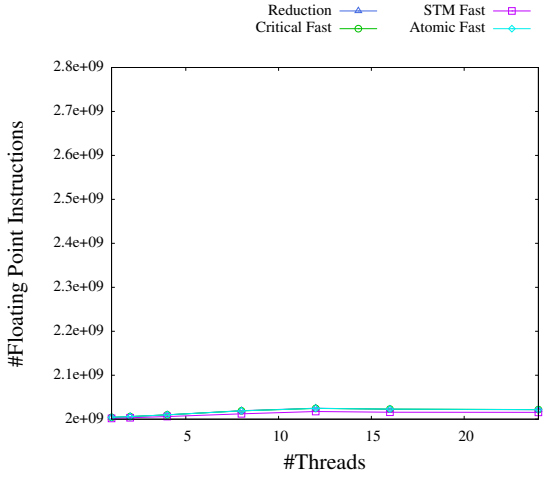
Figure 6.19(c) and Figure 6.19(d) illustrate the number of floating point instructions executed on the y-axis over the number of threads on the x-axis for normal and pipelined CG respectively. Across all thread counts, the number of floating point instructions is stable (or subject to marginal changes as with *STM Fast* and pipelined CG). Pipelined CG requires 25 % more floating point instructions than normal CG to solve the same problem of which only 4 % are due to a changed convergence behavior. Thus, the different algorithmic structure has a large influence on the necessary computation. This finding matches the goal



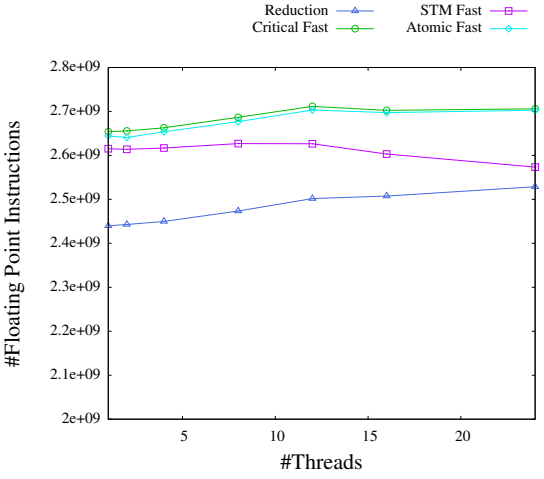
(a) Instructions retired with normal CG.



(b) Instructions retired with pipelined CG.



(c) Floating point instructions in normal CG.

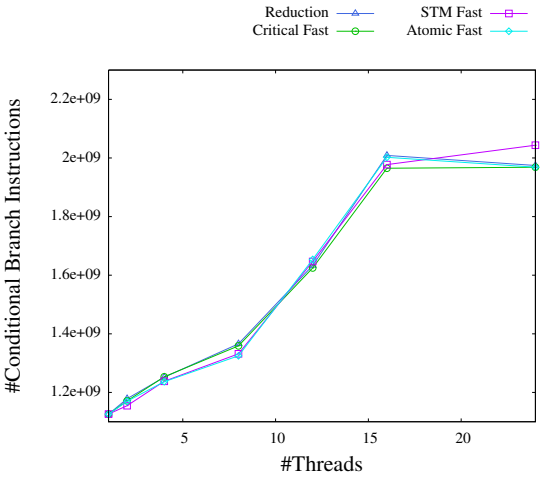


(d) Floating point instructions in pipelined CG.

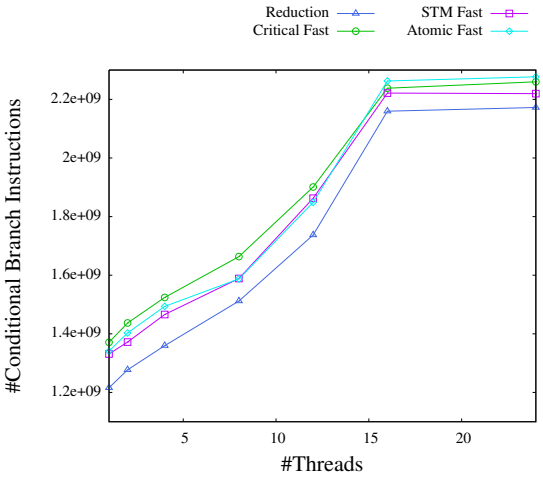
Figure 6.19: Instructions retired and floating point instructions executed on ExpX5670.

of pipelined CG to trade additional computations for a relaxed algorithm that potentially enables the streaming of a vector. Moreover, pipelined CG also shows differences in the synchronization mechanisms such that *Reduction* executes less FP instructions than *STM Fast* which requires less FP instructions than *Critical Fast* and *Atomic Fast*.

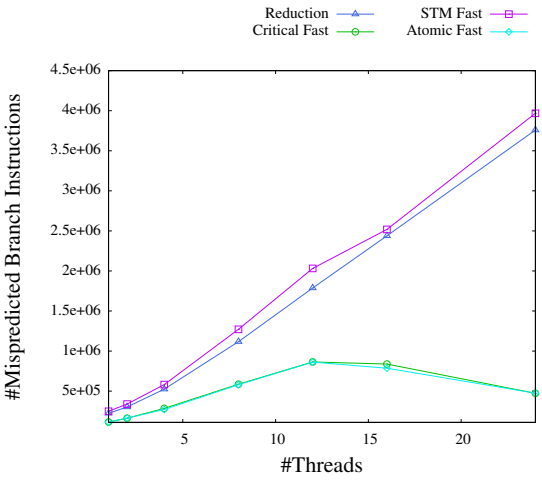
Branch instructions and their prediction is also of importance for the overall performance of the application. Figure 6.20(a) and Figure 6.20(b) show the number of conditional branch instructions executed over the number of threads for normal CG and pipelined CG respectively. Both figures show that all synchronization variants perform a similar number of conditional branches which increases with the number of threads. Except for normal CG with 24 threads, *STM Fast* does not perform more branches than the other synchronization variants. For the performance of the application, the number of mispredicted branches is of great importance because each mispredicted branch comes with a penalty (e.g., through a flush of the processor pipeline). Therefore, Figure 6.20(c) and Figure 6.20(d) exhibit the number of mispredicted branches on y-axis and the number of threads on the x-axis for normal and pipelined CG. In both cases *STM Fast* and *Reduction* show an increasing number of mispredictions as the number of threads increases whereas *Critical Fast* and *Atomic*



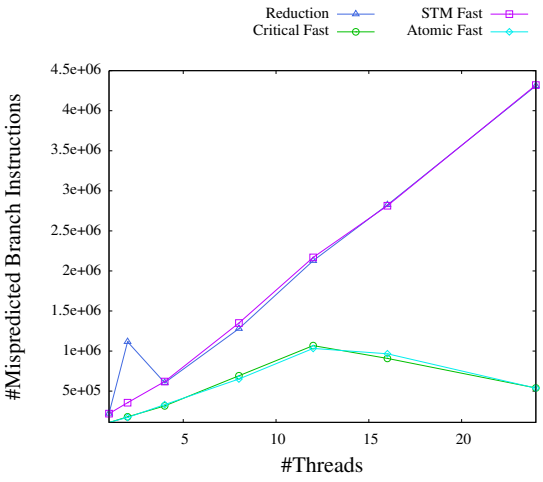
(a) Conditional branch instructions of normal CG



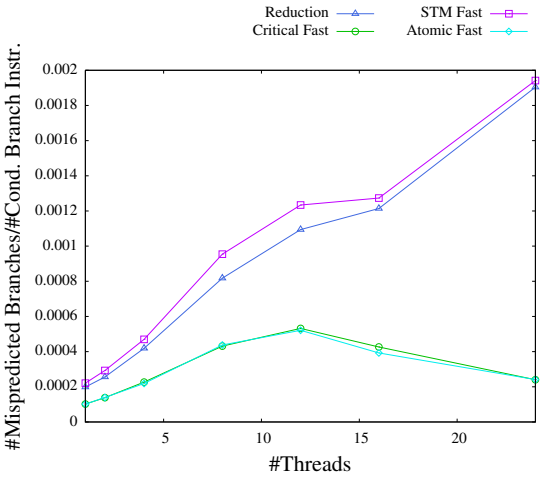
(b) Conditional branch instructions of pipelined CG



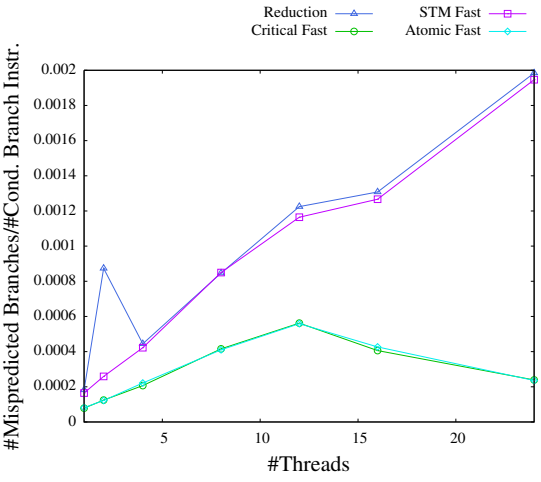
(c) Mispredicted branches of normal CG



(d) Mispredicted branches of pipelined CG

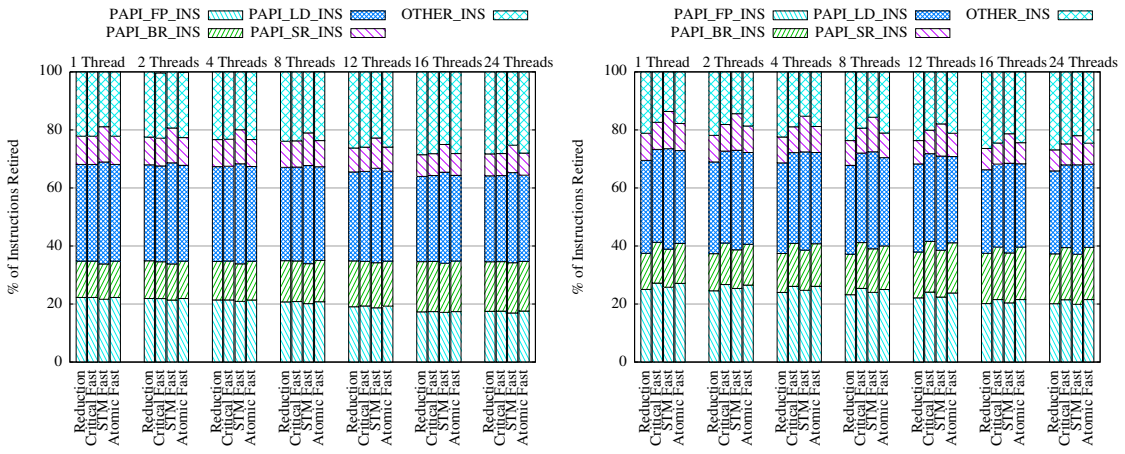


(e) Rate of mispredicted branch instructions for normal CG



(f) Rate of mispredicted branch instructions for pipelined CG

Figure 6.20: CG with conditional branch and mispredicted branch instructions on ExpX5670.



(a) Normal CG

(b) Pipelined CG

Figure 6.21: Breaking down the total amount of instructions in CG on ExpX5670. Figure similar to [176].

Fast reach the peak at 12 threads and then the number of mispredictions decreases as the number of threads increases further. It should be noted that the number of mispredicted branches is not significantly higher for any of the synchronization patterns or one of the algorithmic variants. The relative number of mispredicted branches for all synchronization patterns, algorithmic variations and threads counts is below $0.2\,\%$. This excellent value is due to the regular structure of the CG method that is simple to predict for the branch prediction unit (cf. to Figure 6.20(e) and Figure 6.20(f)).

Which instruction type contributes the largest share? Figure 6.21 shows a breakdown of instructions retired into the measured type of instructions. The available types are: floating point instructions (denoted as PAPI_FP_INS), branch instructions (labeled PAPI_BR_INS), load and store instructions (PAPI_LD_INS and PAPI_SR_INS respectively) and remaining instruction with the label OTHER_INS. Other instructions have not been measured but computed as the difference from the measured ones with the remainder of the retired instructions (PAPI_TOT_INS). The Figure has a normalized y-axis that shows $100\,\%$ of the retired instructions. Each of the instruction types has a box that represents its share of the retired instructions. These bars are grouped according to the thread count and each group shows the used synchronization mechanisms (*Reduction*, *Critical Fast*, *STM Fast* and *Atomic Fast*). The number of threads for each group is also found below the label. Figure 6.21(a) shows the breakdown for normal CG and Figure 6.21(b) shows pipelined CG. For both the following trends can be derived: the share of floating point instructions decreases as the number of threads increases although the actual number of floating point instruction is constant. This is due to the fact that the number of other instructions increases as the number of threads increases. These additional instructions stem from the spawning/coordinating more threads. For normal CG and *Reduction* the share of FP instructions decreases from $22\,\%$ for 1 thread down to $17\,\%$ for 16 threads. Pipelined CG and *Reduction* yields similar numbers: the FP rate decreases from $25\,\%$ for 1 thread to $20\,\%$ for 16 threads. A similar trend can be noted for the number of loads and stores: for *Reduction* the share decreases from $43\,\%$ for 1 thread to $37\,\%$ for 16 threads for normal CG and from $41\,\%$ for 1 thread to $36\,\%$ for 16 threads for pipelined CG. Further, the different share for *STM Fast* deserves mentioning: for normal CG the loads and stores decrease

from 47 % for 1 thread to 41 % for 16 threads (with identical values for pipelined CG). The share of branch instructions increases: for pipelined CG and *Reduction* it increases from 12 % with 1 thread to 17 % with 16 threads and for normal CG it increases from 13 % with 1 thread to 17 % with 16 threads. *STM Fast* shows a similar increase. On the other hand the share of other instructions increases for normal CG and *Reduction* from 22 % (1 thread) to 29 % (with 16 threads) and for pipelined CG from 21 % for 1 thread to 27 % for 16 threads. For *STM Fast* the increase is from 19 % (1 thread) to 25 % (16 threads) for normal CG and from 14 % (1 thread) to 21 % (16 threads) for pipelined CG.

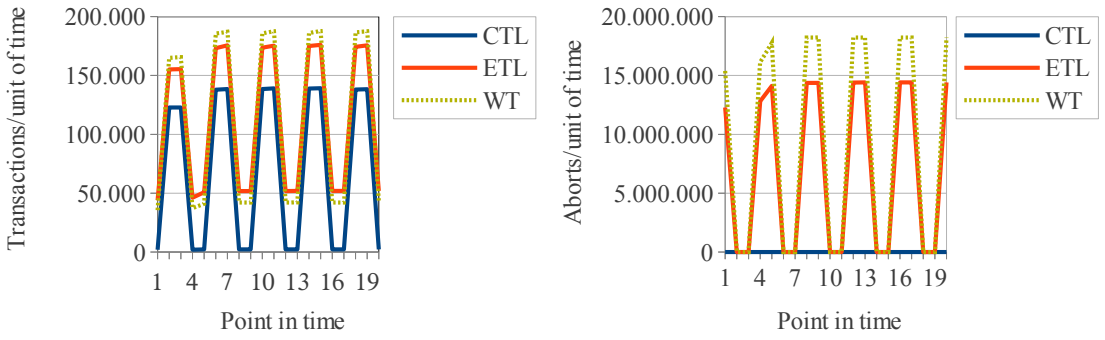
To summarize these findings, we conclude that loads and stores contribute the highest share to the retired instructions ($\approx 40\,\%$), floating point instructions contribute $\approx 20\,\%$ and branch instructions $\approx 15\,\%$ while other instructions contribute $\approx 25\,\%$ and become increasingly important with larger thread counts. The actual numbers for the synchronization variants, thread count and algorithmic choice may vary, but this general trend holds. Therefore, the dominant instructions for CG in the examined setting are loads and stores.

6.3.3 Findings with Normal and Pipelined CG

Our first finding is that the right way of organizing the reductions is the key to performance. A reduction implemented with direct updates of the shared variable, as seen in the *Slow* synchronization variants, will not yield a speedup over execution with one thread regardless of the synchronization primitive. Instead thread-local variables that hold intermediate results, as demonstrated with *Fast* synchronization variants, are a requirement to achieve speedups.

Moreover, the pipelined CG with larger transactions is a strong competitor for normal CG because the number of aborts is modest up to 16 threads. As a downside, pipelined CG required one more iteration to achieve convergence compared with normal CG for our example case. For both CG variants, the wait time at the barriers dominates the time for synchronization in the reduction operations of the *Fast* variants. This also undermines the gains of the parallel execution as well as those due to the optimization of TM. The regular problem structure of CG demands that barriers synchronize all threads after a step in the loop. Thus, a thread that executes a transaction and forces another thread to abort and execute again, simply waits longer at the next barrier for the remaining threads. This basic scenario still holds for longer transactions with pipelined CG. As a result, the CG algorithm is not suited to demonstrate a performance gain with STM. On the other hand, the competitive execution time of pipelined CG with larger transactions and still moderate contention confirms the basic idea of optimizing the TM behavior through employing larger transactions.

The large difference in execution time for transactions and barriers suggests that future research should target more efficient barrier synchronization or techniques to elide barriers. Common to both CG variants, we found that higher thread counts lead to more L2 cache misses that hinder the scalability and that loads and stores contribute the largest amount to all kinds of instructions retired.



(a) Sampling the throughput in transactions per unit of time of three different STM strategies.

(b) STM strategies with aborts per unit of time.

Figure 6.22: Comparison of three STM strategies executing an application which executes either read-only or write-only transactions during a fixed time interval.

6.4 Phase Detection in TM Applications

In contrast to traditional mutual exclusion locks, TM follows an optimistic approach and concurrently executes transactions. To track the memory accesses of different transactions to the same data structures, a run time system (e.g., STM) detects and resolves these conflicts dynamically. Researchers proposed a large variety of STM algorithms for conflict detection, data versioning, and contention management. However, even a simple word-based Software Transactional Memory system (e.g. TinySTM [62]) forces the programmer to decide

- when conflicts are detected (early with encounter-time locking or late with commit-time locking),
- where speculative data is stored (in-place also called write-through or in a buffer also called write-back),
- whether reads are visible to writers or invisible.

The first two design decisions covering the conflict detection and the buffering of transactional data can be combined to three STM strategies. We denote these strategies as ETL (write-back with encounter time locking), CTL (write-back with commit-time locking) and WT (write-through with encounter time locking). The performance of a TM application depends on the choice of the STM strategy: workloads with a small abort rate perform better with an optimistic strategy (e.g., WT or CTL). In particular, data versioning in memory (write-through) features fast commit operations and slow aborts. On the other hand, applications with high abort rates favor a pessimistic strategy (e.g., ETL). Figure 6.22 illustrates these performance differences generated with a synthetic TM application. This TM application is run for 10 seconds and internal counters track successfully completed transactions and aborts. These counters are read and reset at fixed intervals of 500 ms. Each second, the TM application transitions between the execution of read-only and write-only transactions. While the former transactions do not conflict, the latter suffer from writing to exactly the same memory locations in the same order. Figure 6.22(a) holds the throughput of the STM system in transactions per unit of time on the y-axis. The alternating execution of read-only and write-only transactions manifests in peaks and local minima of

the throughput. The Aborts per unit of time (plotted in Figure 6.22(b)) complement the picture. Although all STM strategies reveal a similar application's behavior, performance differences become apparent. Subsequently, a throughput-oriented STM would employ the WT strategy for the read-only and ETL for the write-only phases. However, an STM with the objective to minimize the number of aborts would settle for the CTL strategy (cf. to Figure 6.22(b)).

Although we created the example application artificially, we postulate that TM applications exhibit phase behavior. In order to systematically investigate the phase behavior of TM applications, this section introduces two novel algorithms for phase detection in TM applications [72] that are integrated in the VisOTMA framework. The components, alltogether forming the *Transactional Memory Phase Detector (TMPD)*, enable a programmer to systematically investigate the TM application's behavior, using a compelling user interface, and help to make a profound decision which STM strategy to select. In contrast to previous work that selects one strategy for the whole program run, our approach detects program phases and, thus, uncovers further optimization potential. This section contributes the first algorithms for offline phase detection in TM applications to the state of the art.

6.4.1 Comparison with Related Work

A survey of phase detection techniques for sequential and parallel applications is given in Section 3.8. These approaches do not cover TM applications yet. The goal of detecting phases in TM applications is to have a runtime system that can adapt to these phases and show a shorter run time or lower abort rate than before. With this background, related work in the field of adapting transactional memory systems (cf. to Section 3.7) is of great importance. Marathe et al. present an *Adaptive Software Transactional Memory* (ASTM) [131]. ASTM target obstruction-free or lock-free STMs that detect conflicts on object granularity. As two ends of the spectrum, ASTM uses DSTM (eager-acquire) and OSTM (lazy-acquire) that differ in details of the organization of the meta data and progress guarantees. These differences lead to huge performance differences depending on the workload [132]. DSTM favors write-dominated workloads (and performs significantly better than OSTM). On read-dominated applications, OSTM performs twice as fast as DSTM. These performance differences motivate ASTM. ASTM lends from both STMs to perform well on both kinds of workloads. Moreover the work also demonstrates that a history helps to adapt the semantics of acquiring an object. The throughput of ASTM is on a par with the best of the considered STMs.

Inspired by these findings and the dramatic gains in performance, our work aims to detect phased executions in TM applications in general in order to trigger the adaptation of an STM, such as ASTM, at a finer granularity. In contrast to the work presented in Section 3.7 that either uses local adaptation of acquire strategies or tunes STM-specific parameters implemented inside the STM system, our approach aims at rating the global behavior of the TM application and providing an approach that is orthogonal to the specific STM system. Through the use of post-processing tools and visualizing the behavior of the application, even an inexperienced programmer can take advantage of our approach. Moreover, considering the global behavior of the TM application also promises larger gains through exploiting the phase changes. Due to the global scope, this approach is not limited to changing only the acquire strategy or other STM intrinsic parameters but also needs to face an additional synchronization of threads for policy changes.

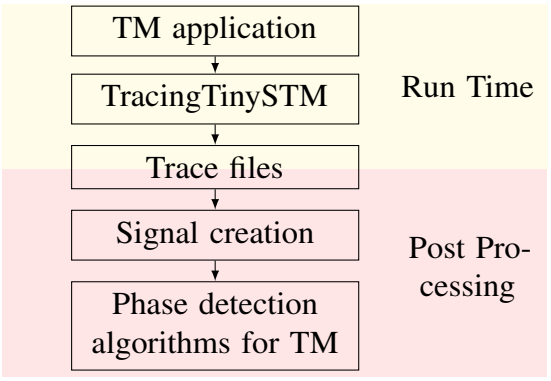


Figure 6.23: Workflow for phase detection in transactional memory applications.

6.4.2 Design of the TM Phase Detector

This section depicts the concept to enable offline phase detection in transactional memory applications. A schematic overview is given in Figure 6.23. The TM application (without modifications) is run on top of an extended STM system. The enhanced TinySTM, now called TracingTinySTM, generates event traces. Chapter 5 describes the design and performance of the TracingTinySTM as well as the information contained in the trace files. The Transactional Memory Phase Detector (TMPD) processes these trace files in a post-processing step. The TMPD incorporates two novel algorithms for phase detection in TM applications. These algorithms dupped *Signal Analysis* and *Wavelet Transform* are introduced in this chapter. In order to make the raw trace files amendable for these algorithms, a first processing step transcodes and aggregates the information from the trace files. The result of this step is a simple aggregate representation that reflects the conflict potential in form of a signal. In the following, we introduce the TMPD components with emphasis on the signal creation and phase detection algorithms.

Creating a Signal from Event Traces This paragraph highlights in the following the generation of the digital signal. To enable signal generation in a transactional memory context, we customized an approach described in the literature [26]. In this previous approach, the considered events are *Running* and *Idle* processor states derived from trace files with time stamps. The methodology is appropriate because the automatic phase detection is desgined for large-scale MPI applications such as weather research and forecasting and applied to e.g., the Nonhydrostatic Mesoscale Model. However, time stamping all transactional events in a multi-threaded environment results in a large overhead. Consequently, we developed a lightweight time-stamping scheme, which allows to reconstruct the total order of transactions without introducing an additional synchronization point. The key is the combination of two time-stamping mechanisms: a heavy-weight operating system time stamp that is synchronized across all processor cores and a light-weight core-specific cycle counter (TSC). While the former is only taken at the startup of a thread, the latter is taken during startup and at all transactional start, abort and commit actions, though not at reads and writes. Due to the synchronized OS time stamp, an absolute ordering of events is possible by calculating the relative number of cycles which elapsed since thread startup. A calculation transforming the cycles into elapsed time enables us to operate with absolute time stamps. Herewith, two transactions are examined whether their execution times overlap. Moreover, Table 5.1 in Chapter 5 reveals that on each transactional read or write access the data address is logged. We use these addresses in the next step to identify whether two transactions access the same memory or not.

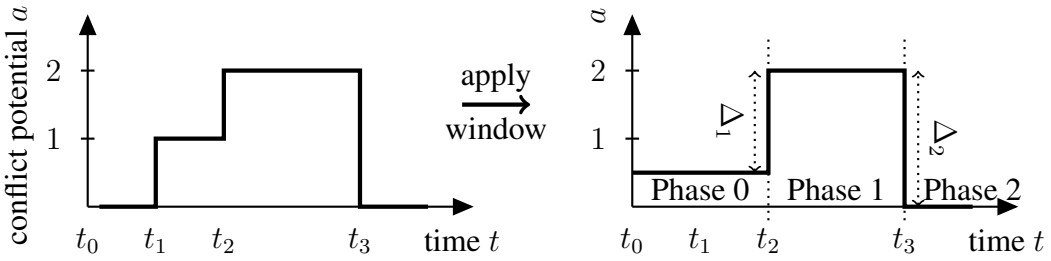


Figure 6.24: Digital Signal created from conflict potential of concurrently running transactions on the left hand side. After applying the window of fixed size, the phase detection is performed.

The signal is calculated according to Equation 6.17. Let $h(t)$ be the amplitude that depends on the conflict potential of the transactions i and j at time t.

$$h_{ij}(t) = \begin{cases} 1, & \text{conflict at time } t \text{ detected} \\ 0, & \text{no conflict is detected or } i = j. \end{cases} \tag{6.16}$$

Two transactions are said to be conflicting if the following criteria are satisfied:

1. the transactions overlap in time and
2. at least one of the transactions writes the data address which the other transaction accesses (reads or writes).

The first point enables to abstract from the actual execution while preserving the characteristic of the application. In particular, we tackle the issue to conclude from the trace files, generated during one determinate execution of the application, to a general phase behavior. Especially when considering TM workloads, the OS scheduling impacts performance: two threads A and B scheduled together may create a conflict if A starts a transaction before B commits the ongoing transaction e.g., because both write the same address. Whereas delaying the start of the transaction in thread A until B is about to commit, does not result in a conflict. In TM, this means that in the first case a transaction is aborted and executed again whereas in the second case both transactions run without conflict. From the standpoint of the conflict probability both cases are equal because whether or not a conflict will manifest during the next execution depends on the operating system. We attempt to mitigate this issue by comparing all memory accesses (reads and writes) of overlapping transactions. In case we detect a conflict inducing access pattern to the same address, the conflict potential for these two transactions increases by one. Further conflicts of these two transactions are not accounted for. In the following, we will formalize this method. The inspection of the two overlapping transactions produces h_{ij} for transactions i and j as defined earlier in Equation 6.16. For each point in time t, the amplitude of the digital signal a is calculated through summing up over h_{ij}:

$$a(t) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} h_{ij}(t). \tag{6.17}$$

In order to clarify this procedure, we show an example with three transactions. The left hand side of Figure 6.24 illustrates two transactions that conflict at time t_1. Then, a third

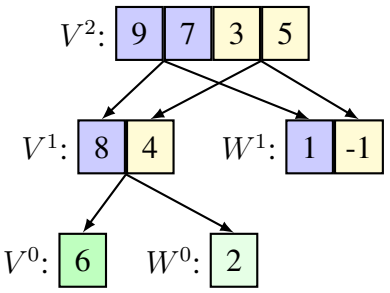


Figure 6.25: Haar wavelet transform of the digital signal [196].

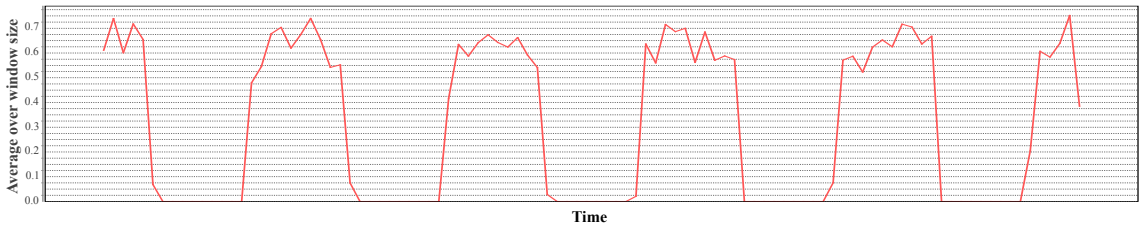
transaction also has a conflict with one of the previous transactions at time t_2 – as a consequence, the signal rises and then drops at t_3. At t_3, all transactions are finished such that no conflict potential remains: the signal is zero.

The amplitude range of the signal depends on the number of threads n: $a_{max} = \frac{(n*(n-1))}{2}$, with a_{max} being the maximal amplitude of the digital signal. This digital signal is used as the basis for two phase detection algorithms: *Signal Analysis* and *Wavelet Transform*.

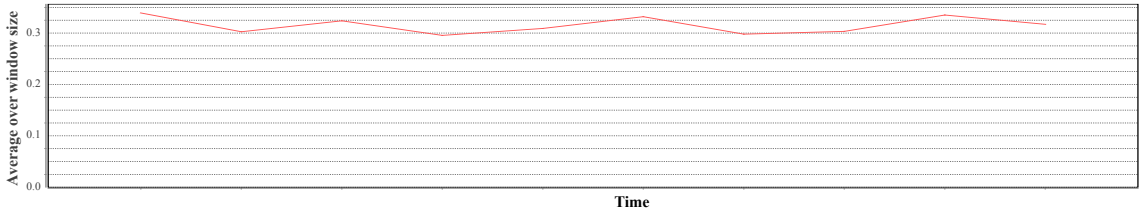
Signal Analysis Once the digital signal, representing the conflict potential of transactions, is created, the signal analysis algorithm proceeds. The algorithm first divides the signal into sections of fixed size (called windows). The window size is part of the algorithm's parameter set and can be specified by the user. Varying the window size enables the detection of program phases of differing lengths. For each section, the arithmetic mean signal is calculated and stored in an array. For subsequent sections, the mean values are compared. If the difference exceeds the user-defined threshold th, a phase change is detected and reported. Obviously a phase detection procedure requires to set th such that $th \in]0, a_{max}[$. With $th = 0$, every signal change results in a detected phase whereas $th = a_{max}$ will not yield any detected phase changes. However, a meaningful value for th is to be determined experimentally.

The right hand side of Figure 6.24 illustrates the detection of phase changes with the algorithm *Signal Analysis*. First we apply a window of size s with $s = t_2 - t_0$. The window evens out the two plateaus between $[t_0, t_1]$ and $[t_1, t_2]$ from the original signal so that the average of the previous plateaus remains in $[t_0, t_2]$. In $[t_2, t_3]$ the averaging window does not change the original signal. Setting the threshold $th = 1$ yields two detected phase changes. For this detection, we compare Δ_1 and Δ_2, that are the differences of the adjacent windows, to the threshold th. The algorithm detects a phase change if $\Delta > th$. Thus, for the simple example, the Signal Analysis algorithm detects two phase changes. In order to exploit these phase changes, the threshold th must be high and the number of detected phase changes should be low. In case th is high, the adjacent program sections have different characteristics. In case the number of phase changes is small, the overhead of changing from one STM system to the next is limited. In order to successfully exploit the phase behavior, both criteria must be satisfied.

Wavelet Transform Wavelets originate from functional analysis and enable the decomposition of functions in coarse and fine/high frequency parts [196]. The coarse part describes the overall shape whereas the level of detail is controlled with the finer parts. This makes wavelets amendable for compression because some of the coefficients that relate to high frequency parts of the image can be left out without causing visible artifacts. Other desired properties of wavelets are that, in contrast to other transformations from



(a) Averaged signal for a window size that is equal to 1 % of the signal length.



(b) Averaged signal for a window size that is equal to 10 % of the signal length.

Figure 6.26: Influence of using the average value of a window with different sizes for the signal analysis algorithm.

the time into the frequency domain like the fourier transform, wavelets are preserving the locality. Therefore, one can match the original signal with its wavelet transformed counter part. Moreover, the wavelet transformation supports multi-resolution so that it supports refinement of an initial transformation if this is desired. Several applications of the wavelet transform have been shown in Chapter 3.8. For our particular goal, we employ the one-dimensional Haar wavelet transformation as it is straightforward to implement and fast. Stollnitz et al. present a thorough introduction to wavelets and their application in computer graphics in [196]. Figure 6.25 illustrates the one-dimensional Haar wavelet transformation. The digital signal is represented as vector V^2. From the first two elements of V^2 the arithmetic mean is stored in V^1. This is the coarse part of the signal obtained through averaging. Further, the difference to this mean is kept in W^1. In a next step V^1 is used to compute V^0 and W^0. The user can specify the number of iterations N (in this example $N = 2$). The vectors W are needed for the lossless reconstruction of the original signal and contain the high frequency parts of the signal obtained through calculating the differences. In computer grafics these are also named detail coefficients. In order to execute a fully fledged phase detection with Haar wavelets, we would have to preserve these detail coefficients and later on use a K-means clustering algorithm to determine how similar two of the resulting vectors are (cf. to [98]). For our purpose here, the high frequency parts are of minor importance because only a TM phase behavior that differs significantly in the coarse grain coefficients will lent itself to being exploitable. Therefore, we omit the high frequency parts and also do not require the K-means clustering. For additional savings in time and space we do not compute the vectors W. With the remaining vector V, adjacent values are compared to a threshold value and a phase change is detected. Thus, after applying this reduced Haar wavelet transform, we can not expect the detected phases to have a different quality than those detected with *Signal Analysis* but the comparison will provide an insight into the relations of the parameter settings for both algorithms.

6.4.3 Applying Phase Detection Algorithms to the STAMP Suite

In this section the results of the phase detection algorithms are evaluated. Starting with the artificial example application from Chapter 1, we illustrate the impact of specific

Name	Input parameter set
bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2
genome	-g256 -s16 -n16384
intruder	-a10 -l4 -n2038 -s1
kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16.txt
labyrinth	-i random-x32-y32-z3-n96.txt
ssca2	-s13 -i1.0 -u1.0 -l3 -p3
vacation-low	-n2 -q90 -u98 -r16384 -t4096
vacation-high	-n4 -q60 -u90 -r16384 -t4096
yada	-a20 -i yada/inputs/633.2

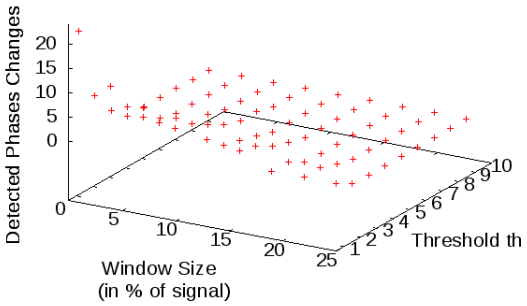
Table 6.3: STAMP input parameter sets used for evaluation of phase detection algorithms.

algorithmic parameters. With parameters deduced from the findings in this small example, we evaluate the STAMP benchmark suite which contains widely-accepted transactional memory benchmarks [24].

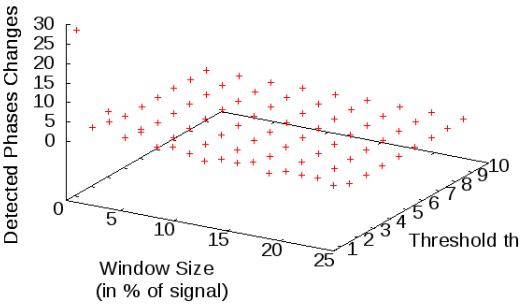
Parameter choice. The detection of program phases depends on the parameter setting of the algorithm. In the following example, we study the impact of setting the window size of the algorithm signal analysis to $1\,\%$ and $10\,\%$. A larger window size leads to a larger amount of averaged values of the signal, thus, reducing the influence of a single value. When weighting each value equally, frequent phase changes with small duration may occur. Figure 6.26 highlights the impact on the resulting signal. The x-axis represents the time whereas the y-axis holds the conflict potential of the transactions. Please recall that the application is by construction changing between two modes of execution: running conflicting or read-only transactions. When setting the parameter window size to $1\,\%$ (Figure 6.26(a)) this behavior is clearly preserved. The plots show two threads. The conflict potential transitions between 0.7 (with conflicts) and 0 (without conflicts). Thus, the application behavior is very well preserved. However, when setting the window size to $10\,\%$ (cf. to Figure 6.26(b)), the signal ranges from 0.3 to 0.35 and a phase change can not be detected. Thus, the latter parameter setting is not adequate to capture the phase behavior for this example.

The STAMP benchmark suite contains eight benchmarks equipped with different input data sets. Two programs (kmeans and vacation) have input data set for different transactional behavior (high or low contention). Each benchmark executes with 8 threads on Exp2378 and runs to completion with the small input sizes. The parameter sets for the benchmarks are listed in Table 6.3. Trace files, generated during the execution, are then post-processed by the phase detection algorithms.

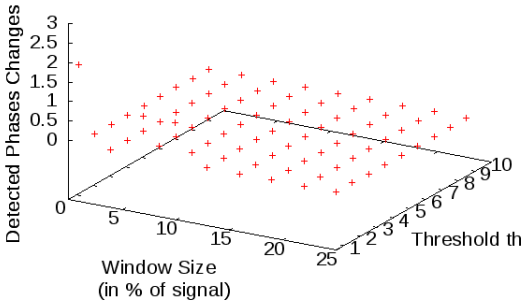
Figure 6.27 depicts the phase changes detected in the STAMP applications when using the algorithm Signal Analysis. The parameter space of the algorithm is explored: the threshold ($th \in (1, 10)$) as well as the window size (($1\,\%, 25\,\%$) of the signal size) are varied. These ranges are well within the theoretical limits ($0 < th < a = 27$) and guided by the findings in the last section. The z-axis depicts the number of detected phase changes. Bayes (in Figure 6.27(a)), intruder (in Figure 6.27(d)), kmeans with high contention (in Figure 6.27(e)), kmeans with low contention (in Figure 6.27(f)), ssca2 (in Figure 6.27(g)) and yada (in Figure 6.27(i)) show phase changes (the actual number ranges from 2 to 28) for small values of th and small to middle sized windows.



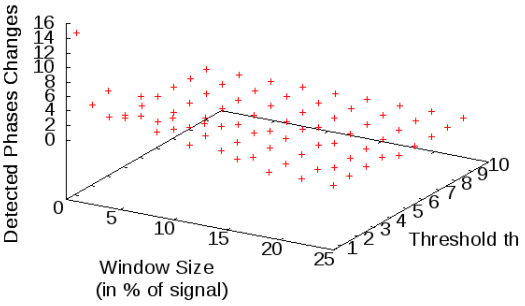
(a) `Bayes` benchmark.



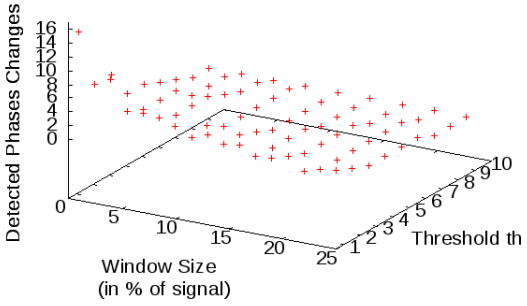
(b) `Labyrinth` benchmark.



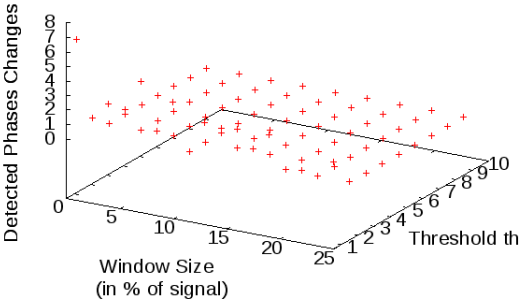
(c) `Genome` benchmark.



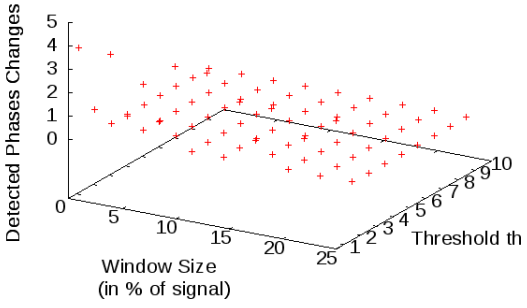
(d) `Intruder` benchmark.



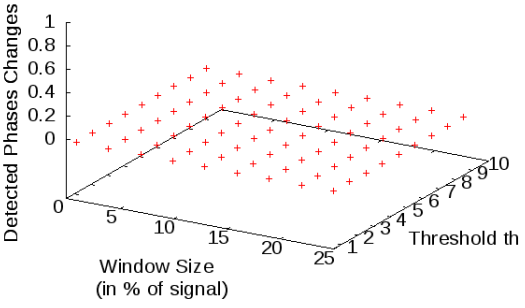
(e) `Kmeans` with high contention.



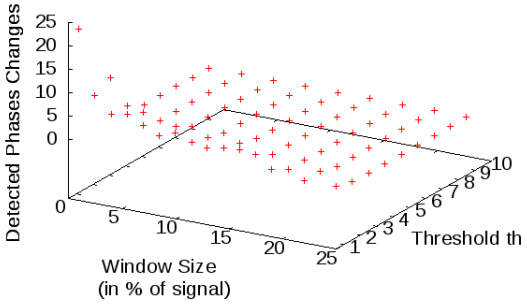
(f) `Kmeans` with low contention.



(g) `Ssca2` benchmark.



(h) `Vacation` with high contention.



(i) `Yada` benchmark.

Figure 6.27: Phase Detection with Signal Analysis of STAMP benchmarks.

`Labyrinth` (cf. to Figure 6.27(b)) and `genome` (cf. to Figure 6.27(c)) show phase changes only for small values of th and small window sizes. For `vacation` (cf. to Figure 6.27(h) with both versions - one without plot) no phases changes are detected with *Signal Analysis*.

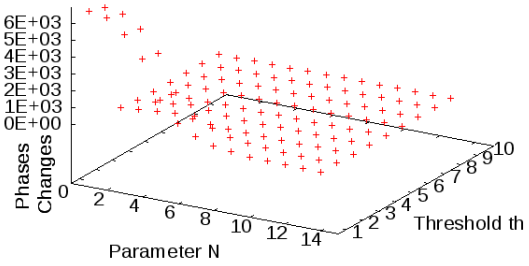
In Figure 6.28 the Wavelet Transform is applied to trace files of the STAMP benchmarks. The parameter setting includes N, the number of iterations, and th. Due to the smaller window sizes for small values of N (window equals 2^N), substantially more phase changes are detected for small values of th. Even both version of vacation (cf. to Figure 6.28(h) and Figure 6.28(i)) exhibit phase changes, although only up to 4 with low contention. Compared to this small maximum number, the phase changes detected with `ssca2` (cf. to Figure 6.28(g)), `kmeans` with high (shown in Figure 6.28(e)) and low contention (cf. to Figure 6.28(f)), `intruder` (cf. to Figure 6.28(d)) and `bayes` (cf. to Figure 6.28(a)) are high (up to $2 * 10^6$). `Labyrinth` (cf. to Figure 6.28(b)) and `genome` (cf. to Figure 6.28(c)) show up to 200 phase changes. Phase detection with `yada` did not finish in due time. All detected phases with the *Wavelet* method have in common that the threshold value th must be very small ($\approx 1 - 2$).

Performance Gains. As a last step, we would like to draw the attention to the possible performance gains from exploiting phases. In the synthetic example from Figure 6.22, the phase changes are visible and we can calculate the highest possible throughput. The assumption is a zero overhead mechanism that selects the strategy with the highest throughput (in transactions per second) at the sampling points. The selection alternates between a write-through strategy (WT) for read-only transactions and ETL (encounter-time locking with write back) for writing transactions. Compared to the overall throughput in transactions per second of the WT strategy (highest), the improvement is $4.3\,\%$. If the programmer selects an STM strategy at random, the resulting performance equals the average throughput of all three strategies. Thus, compared to the average throughput of the three strategies, the proposed combined strategy achieves a $20\,\%$ improvement in transactions per second for this synthetic example. These numbers are the theoretical maximum and will be hard to achieve in practice.

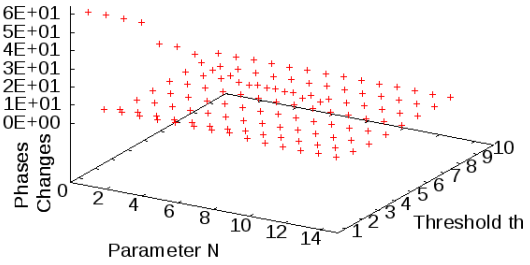
Especially the results from the previous paragraph demonstrate that phase changes in practice are not as pronounced as in the synthetic example. Otherwise, we would detect phase changes for higher values of the threshhold th. Moreover the phases are short - with longer phases, the detection of phases would also report changes for larger window sizes (or higher values of N for Wavelets). Regarding both findings together, we draw the conclusion that successfully exploiting the discovered TM phase behavior through changing the STM strategy for a blocking STM appears to be infeasible. The reason is that the phases are not long enough and the similarity of these phases is high. Thus, a blocking STM system does not show a tremendous performance difference and the overhead of switching the STM strategy will not amortize over time.

6.4.4 Discussion of Phase Detection for TM

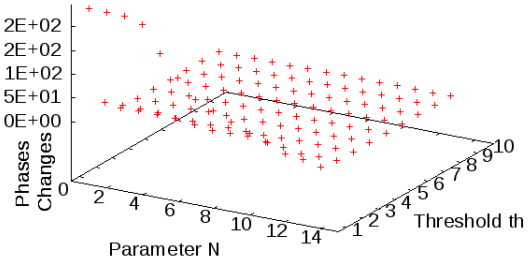
In this previous section, we present a systematic approach to detect phase behavior in transactional memory applications that is integrated in the VisOTMA framework. We introduce the Transactional Memory Phase Detector (TMPD) together with two adapted phase detection algorithms: *Signal Analysis* and *Wavelet Transform*. The results show that we succeed to identify phase behavior in an artificial show case as well as in the STAMP



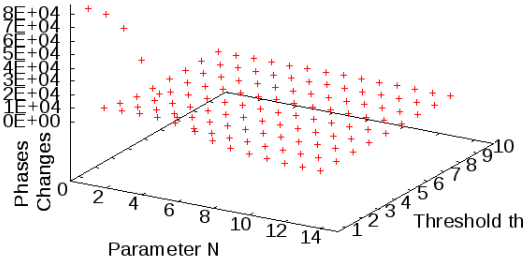
(a) `Bayes` benchmark.



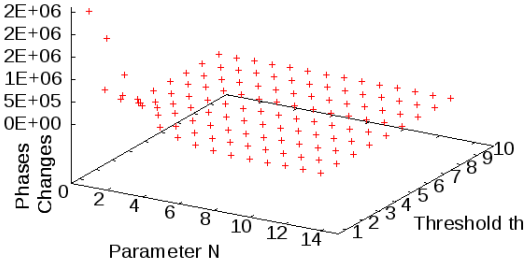
(b) `Labyrinth` benchmark.



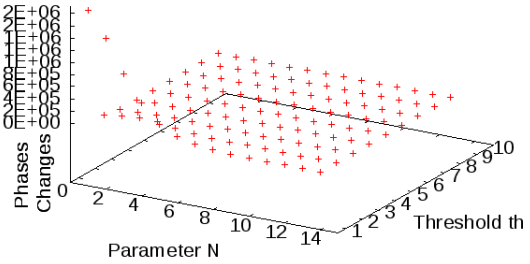
(c) `Genome` benchmark.



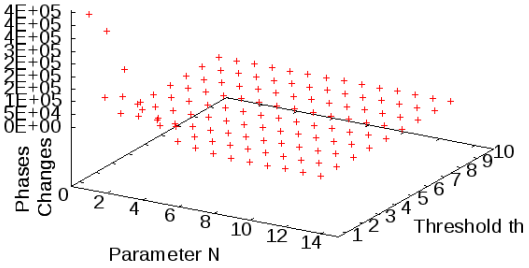
(d) `Intruder` benchmark.



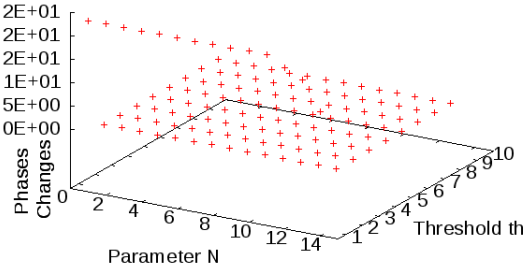
(e) `Kmeans` with high contention.



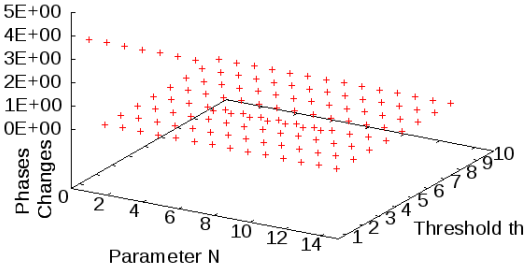
(f) `Kmeans` with low contention.



(g) `Ssca2` benchmark.



(h) `Vacation` with high contention.



(i) `Vacation` with low contention.

Figure 6.28: Phase Detection using Haar Wavelets of the STAMP benchmarks.

transactional memory benchmarks and studied the influence of the parameter settings on the phase changes. For the artifical case, a performance gain between $4.3\,\%$ and $20\,\%$ can be estimated if the overhead for changing the STM strategy is neglected. For the practical STAMP benchmarks, changing the STM strategy for the considered blocking, word-based STM appears to be infeasible. The reason is that the phases are not long enough and the similarity of these phases is high. Thus, a blocking STM system does not show a tremendous performance difference and the overhead of switching the STM strategy, that also must be considered in a realistic scenario, will not amortize over time.

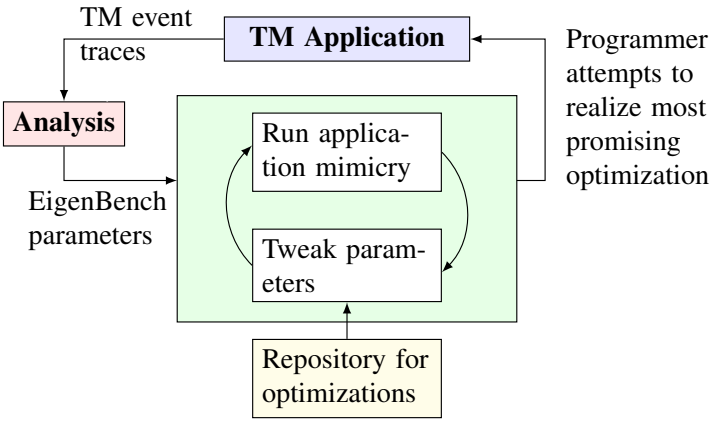


Figure 6.29: Components and interactions to illustrate the optimization workflow with EigenOpt.

6.5 EigenOpt

In this section, we built on the previous idea on using a proxy application, such as PSTMA or CLOMP-TM, to optimize the TM application of interest. Now the focus is on the automation of the optimization process. The idea is to capture the important orthogonal TM characteristics through combining TM events with the readings of hardware performance counters. We use these orthogonal characteristics as input parameters of the EigenBench benchmark [94]. With these parameters the EigenBench benchmark can simulate the TM application behavior. Hence, changes in the parameter set of the benchmark reflect changes in the TM application. We aim to exploit this correlation to simulate optimization attempts through changing the parameters according to a set of optimization patterns and exclude optimization directions with diminishing returns. In case the changed parameter setting reveals a speedup compared with the original one, the optimization must be transferred to the TM application. This may not always be possible, e.g., due to the placement of transactions in the code, but in case the optimization of the TM application succeeds, the performance gains should be similar to the simulated ones. This approach has been described in more detail in [19] and exploits properties of the EigenBench benchmark to optimize a TM application, hence we call the approach *EigenOpt*.

Figure 6.29 illustrates the workflow with EigenOpt. The TM application at the top is not optimized directly. Instead, we generate TM event traces and process these in an analysis phase to extract the parameters for EigenBench. Now the EigenOpt circle starts: The parameters can now be tweaked to resemble an optimization of the application. The resulting application mimicry with EigenBench is run and evaluated. In case the performance is better than the original application, the parameters may be tweaked further. In case a substantial performance gain has been achieved in the simulation, the programmer attempts to realize the optimization with the TM application. The far goal of this approach would be to have a repository for optimizations that have been applied successfully in the past combined with an illustration of the respective programming pattern.

6.5.1 Parameters of Eigenbench

In the following, we will briefly highlight the necessary parameters for the EigenBench benchmark to describe the TM application's behavior in a sufficient level of detail [94].

Name	Meaning	Name	Meaning
N	Number of threads	R_{3i}	Reads of cold array inside Txn
S	Random seed	W_{3i}	Writes of cold array inside Txn
tid	Thread id	R_{3o}	Reads of cold array between Txns
$loops$	Number of Txns per thread	W_{3o}	Writes of cold array between Txns
A_1	Size of Array1 (hot array)	Nop_i	No-ops between TM accesses
A_2	Size of Array2 (mild array)	Nop_o	No-ops outside Txn
A_3	Size of Array3 (cold array)	K_i	Scaler for in-Txn local ops
R_1	Reads/Txn of hot array	K_o	Scaler for out-Txn local ops
W_1	Writes/Txn of hot array	$persist$	Restore random seed if violated
R_2	Reads/Txn of mild array	lct	Probability of address repetition
W_2	Writes/Txn of mild array		

Table 6.4: Parameters used in EigenBench. Taken from [94].

Characteristic	**Definition (Eigenbench Parameters)**
Concurrency	Number of concurrently running threads (N)
Working set size	Size of frequently used memory ($A_1 + A_2 + A_3$)
Transaction length	Number of shared accesses per transaction ($T_{len} = R_1 + R_2 + W_1 + W_2$)
Pollution	Fraction of shared writes to shared accesses ($(W_1 + W_2)/T_{len}$)
Temporal locality	Probability of repeated address per shared access (lct)
Contention	Probability of conflict of a transaction (see Equation (1) [94])
Predominance	Fraction of total shared cycles to total execution cycles ($T_{len} * \alpha/(T_{len} * \alpha + C_{in} + C_{out})$)
Density	Fraction of non-shared cycles executed outside of transactions to total non-shared cycles ($C_{out}/(C_{in} + C_{out})$)

Table 6.5: Orthogonal TM characteristics; similar to [94].

These EigenBench parameters have been selected so that they are orthogonal and are independent of each other - analogue to orthogonal vectors in linear algebra. Table 6.4 holds these parameters for EigenBench and an explanation. The benchmark uses three arrays A_1, A_2 and A_3 to generate a characteristic TM application's behavior according to the settings of the parameters. While A_1 is transactionally accessed from all threads and generates contention among the threads, threads in transactions only access disjoint portions of A_2. Hence, A_1 is called cold array whereas A_2 is called mild. Threads access A_3, a cold array, outside of transactions and also disjointly. R_i specifies the number of read accesses to A_i whereas W_i defines the number of write accesses to A_i. Hence, the sum of $R_1 + W_1 + R_2 + W_2$ defines the length of a transaction. The parameter `loops` defines the number of transactions executed per thread in the benchmark.

Table 6.5 describes the orthogonal TM characteristics generated with these parameters. The number of concurrently running threads is simple to understand and measure. The working set size can also be measured through counting the accesses inside and outside of a transaction. The transaction length is already present in the trace files through logging all read and write accesses in transactions. Current TM event traces already contain this information. Other parameters such as accesses to the three arrays are difficult to discern. Moreover instructions/cycles spent inside or outside of transactions are currently not part of the information retrieved. Hence, substantial changes to the tracing machinery

C Identifier	Events
stm_init_thread	TIMING1, TIMING2, RTSC
TM_THREAD_ENTER	NONTXSTART, PAPI1, PAPI2, [INST]
TM_BEGIN	NONTXEND, PAPI1, PAPI2, [INST]
stm_start	START, RTSC, RETURN, PAPISTARTSET
TM_SHARED_READ	–
stm_load()	READ, [RTSC]
TM_SHARED_WRITE	–
stm_store	WRITE, [RTSC]
stm_commit	COMMIT, RTSC, RETURN, PAPI1, PAPI2, [INST]
TM_END:	NONTXSTART, PAPI1, PAPI2, [INST]
TM_RESTART	–
stm_rollback	ABORT, RTSC, PAPI1, PAPI2, [INST]
TM_THREAD_EXIT	NONTXEND, PAPI1, PAPI2, [INST]
stm_exit_thread	TIMING1, TIMING2, RTSC

Table 6.6: Additional event types required for EigenOpt. Compile-time options enable the events in square brackets. Taken from [19].

are necessary in order to enable an analysis that extracts the required parameters for EigenBench.

6.5.2 Changes to the TracingTinySTM

Table 6.6 illustrates all events that the tracing machinery must capture to enable EigenOpt. A new event is required to distinguish between transactional and non-transactional execution. Hence, the start of a non-transactional execution is marked with the NONTXSTART event. The counters track the application during the non-transactional execution that is ended with a NONTXEND event. As expected a NONTXEND event indicates the begin of a transaction so that the counters are read and reset. The reading of the user-defined PAPI events that are complemented with an event that tracks the number of executed instructions (represented through INST). These events enable to distinguish computation inside and outside of transactions.

6.5.3 Adjustments to Post-Processing Tools

The post-processing tools, that use the TM event traces as inputs, must analyze and extract the parameters for the EigenBench microbenchmark. Hence, we register the new events with the parser of the trace files and associate each transaction with non-transactional values that are related to the code before that transaction. The size and the type of accesses to the contentious array A_1 of EigenBench are determined through identifying contentious addresses in the read and write sets. These accesses are counted as R_1 or W_1 respectively. Reads and writes that are not causing contention are accounted to array A_2. To find out how often these addresses are repeated and determine the parameter `lct`, ParaverConvert must divide the number of unique accesses by the number of total accesses per transaction. Hence, the number of unique accesses for each transaction must be computed additionally. Unfortunately, this scheme does not yet account for A_3 with accesses outside of transactions. Hence, these counts must be estimated by other means either through profiling or through counting accesses in the program.

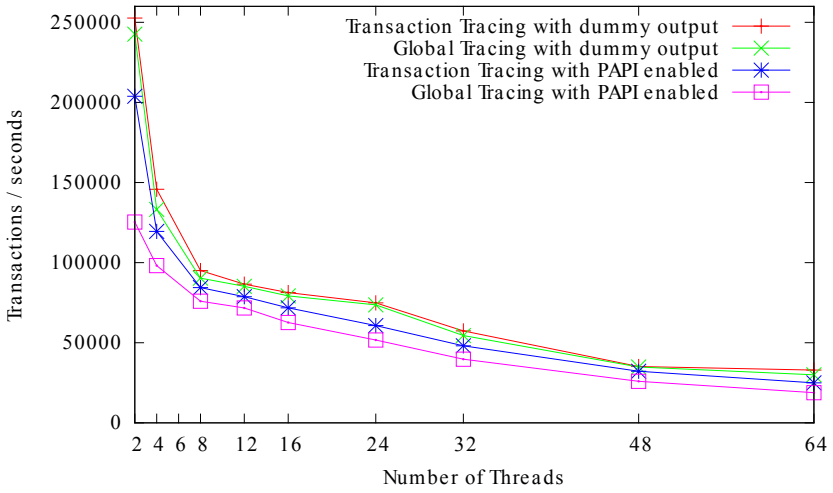


Figure 6.30: Influence of reading PAPI counters on throughput of TM system. Taken from [19].

Due to the lack of experience in the computation of these parameters, the analysis outputs three granularities of aggregated trace data:

- *per atomic block* – gathering statistics for each atomic block defined in the source code separately,
- *per thread* – collecting statistics for each thread separately, and
- *per application* – using averages over multiple transactions and multiple threads.

We expect *per atomic block* to come closest to the behavior of the original application because it enables the highest resolution of parameters. *per thread* averages the parameters of all transactions of a thread whereas *per application* additionally averages the parameters of the threads, hence both metrics should yields more coarse grain parameters. The post-processing tool uses a separate file for each of the three metrics.

6.5.4 Intrusiveness with EigenOpt

The intrusiveness of introducing additional PAPI events and reading the hardware performance counters is quantified in this section. Our experiments aim at distinguishing whether the reading of hardware counters through PAPI introduces the delays and adds to the intrusiveness or the writing of the additional events that are handled by the tracing machinery. Therefore, we compare the two tracing strategies *Transaction Tracing* and *Global Tracing* in two different settings. In the first setting the PAPI interface is used to read and reset the hardware performance counters as described before (called *PAPI enabled*). In the second setting dummy values are passed to the event logging system to mimic the same utilization as with PAPI while at the same time by-passing the overheads associated with reading and resetting the counters. This setting is called *dummy output*.

Figure 6.30 illustrates the differences with the two tracing strategies and clarifies the overhead of PAPI. The y-axis holds the Transactions executed per second. The x-axis shows the number of threads. For both tracing strategies and all threads the figure clearly shows that the dummy output yields a higher throughput in transactions per second. Hence, the influence of the overheads associated with accessing the performance counters

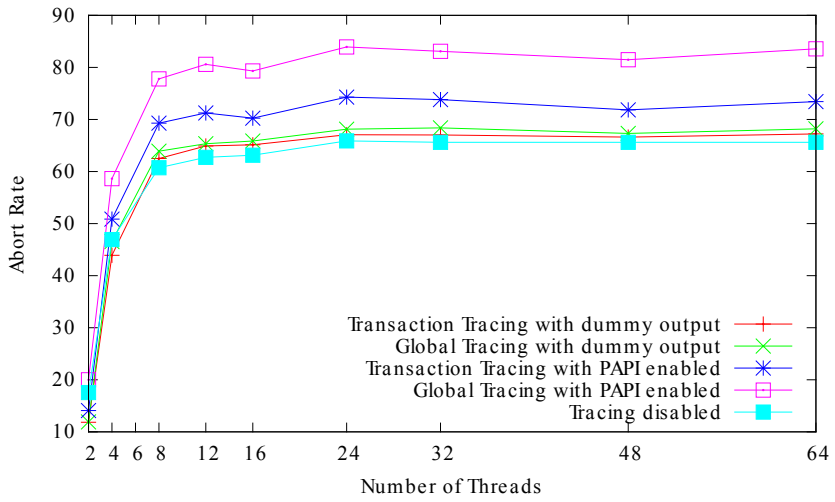


Figure 6.31: Influence of reading PAPI counters on the abort rate. Taken from [19].

through PAPI are significant. Hence, the tracing machinery is not the main source for the intrusiveness but the indispensable use of the hardware performance counters. Moreover, this experiment also reveals that *Transaction Tracing* yields a higher throughput than *Global Tracing*.

As we mentioned before in Chapter 5, a degraded throughput does not necessarily mean a high intrusiveness in terms of altered transactional behavior. Hence, we also compare the abort rate, that is aborts per commit of a transaction, of the four described variants with execution without tracing. Figure 6.31 shows the findings. The y-axis holds the abort rate. The x-axis shows the number of threads. While the deviation with 2 and 4 threads is high, higher thread counts (greater than 8) clearly show that the variant without tracing shows the lowest abort rate. Both strategies with *dummy output*s come fairly close and only show a slightly increased abort rate. Both variants with PAPI enabled show a clearly increased abort rate. Again, *Transaction Tracing* yields a better abort rate than *Global Tracing*.

From the results of this experiments we conclude that accessing PAPI performance counters has a major influence on the recorded TM application behavior. While the tracing machinery is adequate to handle the amounts of trace data and only introduces a small probe effect, the reading and resetting of performance counters is the source of additional overheads and contributes the larger part to the probe effect. Moreover, the strategy of *Transaction Tracing* shows a lower intrusiveness than *Global Tracing*.

Improving the Quality of the EigenBench Parameters

The parameters of EigenBench do not account for the overhead that is associated with using an STM library. When an access to shared memory is carried out by means of an STM library, each shared memory access (e.g., write to A_1) results in multiple memory accesses that do the STM book keeping, e.g. check locks, buffer the speculative value, to provide isolation between transactions. For our apporach this means that we will measure all the instructions carried out by the STM and need to find a way to deduce the original EigenBench parameter. The previous example with a single memory access illustrates that the measured counts are higher than the original parameters. Hence, we need to find a way to account for and reduce this systematic over-estimation.

Iteration	R_{3i}	W_{3i}	Nop_i	R_{3o}	W_{3o}	Nop_o
InTxn	10	10	10	0	0	0
1	1 595	1 067	2 540	266	198	604
2	179 107	116 979	268 518	266	198	604
3	19 809 914	12 845 297	24 753 117	266	198	604
NonTxn	0	0	0	10	10	10
1	270	200	535	1 588	1 063	2 610
2	270	200	535	178 748	116 723	267 699
3	270	200	535	19 765 394	12 916 327	29 687 539

Table 6.7: Cases InTxn and NonTxn with known inputs are used to quantify the parameters that accumulate over repeated runs of the simulation due to the overheads of STM. Taken from [19].

Table 6.7 illustrates two test cases that execute with the original EigenBench parameters shown in line InTxn and NonTxn respectively. A simulated run with these parameters yields the parameters after one iteration. InTxn does not use any non-transactional parameters and sets these to zero for this experiment whereas NonTxn sets the in-transactional parameters to zero. The remaining parameters are transferred from one simulation run to the next such that iteration two uses the parameter set determined with iteration one and so on. Clearly the table illustrates that the free parameters grow super-linearly whereas the fixed ones yield the same values each time. This illustrates on the one hand that for fixed parameters the results are stable and reproducible and on the other hand that the fast growing parameters should be mitigated through a factor. Hence, we determine a factor for each parameter in Table 6.7 that is used as a divisor to enable reproducible results with this kind of iterative experiments. These divisors are output into a separate configuration file in order to reduce the overheads accumulated in the parameter sets. Of course this simple scheme does not work equally well for all cases but at least it approximates the original and the measured input parameters as further experiments with mixed workloads reveal [19].

Characteristic	`Intruder`	*per application*	*per atomic block*	*per thread*
Transaction Percentage	65.3%	88.4%	83.6%	89.6%
Transaction Size	4 460	10 900	8 860	9 890
Read Set Size Ratio	95.5%	84.1%	67.9%	84.1%
Write Set Size Ratio	99.9%	96.7%	74%	96.7%
Read Set Conflict Density	0.4%	0.02%	0.003%	0.03%
Write Set Conflict Density	5.3%	0.3%	0.006%	0.4%
Read Set Size	15.9	16.8	2.93	16.8
Write Set Size	1.58	0.994	0.43	0.996
Write Read Ratio	9.21%	5.53%	1.36%	5.54%

Table 6.8: Transactional characteristics of the `intruder` benchmark executed with four threads on ExpQ6600. Similar to [19].

Parameter	Case1	Case1 with two transactions
A_1	30 938	30 938
A_2	32 000	32 000
R_1	29	15
W_1	11	6
R_2	99	50
W_2	38	19
lct	48	48
$loops$	25 000	50 000
R_{3i}	0	0
W_{3i}	0	0
Nop_i	0	0
R_{3o}	73	35
W_{3o}	21	10
Nop_o	51	25

Table 6.9: Parameters obtained from traces of Case1 with one long transaction and simulation of adjusted parameters with two shorter transactions. Executed with 100 000 transactions and four threads on ExpQ6600 using the *per application* metric. Similar to [19].

6.5.5 Results with EigenOpt

Intruder Benchmark

As a first benchmark, we use the `intruder` benchmark from the STAMP suite [24]. This benchmarks helps to see whether our approach is already fit for more complex and real-life applications. In order to compare the execution and the measured parameters, we use TM characteristics from [100] to describe the TM behavior. The overheads have been removed from the measured values with the techniques described in Section 6.5.4. Table 6.8 holds the original parameters and the measured parameters *per application*, *per thread*, and *per atomic block*. Obviously, the *per atomic block* metric yields parameters that deviate the most from the original parameters. While the *per application* and *per thread* metrics are better, these are still not close. For the read and write set conflict density both metrics deviate by more than a factor of 10. With these large differences from the original parameters, there is no value in trying to optimize the `intruder` benchmark because the mechanically retrieved parameters do not resemble the original application's behavior close enough. Hence, we test a simple test case in the next section.

Test Application

In order to demonstrate the usefullness of this approach we use a simple test application that updates a fixed number of consecutive memory locations in parallel (cf. Figure 6.7 in [19]). In this application, from hereon called *Case1*, the application first reads all and later writes these locations inside the same transaction. Further we add dummy computation inside and outside of transactions to increase in-transaction times and reduce the pressure through repeated execution of transactions.

Table 6.9 holds the original and simulated characteristics of Case1 and illustrates that both are close enough so that the simulation resembles the original execution. Hence,

we proceed by setting up two scenarios with different contention levels in the following. Our findings show that a higher abort rate must not mean a longer execution time but also illustrate that even the simple question whether to use one long or rather two short transactions may have two different answers depending on the contention level. Hence, simulation helps to find out whether a change is profitable or not. In the following, we focus on the results whereas a detailed description of the process with additional parameter sets and an additional example is presented in [19].

Table 6.10 holds the performance data that represents the first optimization attempt. Case1 shows the initial performance of the test application with a high abort rate. A possible optimization attempt is to split the large transaction into two smaller ones, herewidth reducing the size of the read and write sets in the STM and the execution time of a transaction. A shorter execution time reduces the likehood of a conflict while a smaller read and write sets reduce the amount of wasted work in case a transaction rolls back. Case1 with two transactions illustrates the changed application's behavior and shows a reduced execution time and a smaller abort rate. The speedup in execution time due to the two transactions is 2.65 x with respect to the original case with only one transaction. Of course this large performance gain may not be generalized but the small test illustrates that significant improvements are possible. This example shows that an optimization that has been simulated first with the parameters of the EigenBench microbenchmark and its execution, may be transferred to the original application and yield performance gains.

Table 6.11 demonstrates that these optimization are difficult to carry out without simulation. The very same Case1 with a different parameter setting that increases the number of memory locations by a factor of 62.5 yields a decreased abort rate for the original case and the case with two transactions is slower although it yields a lower abort rate. Here the decreased data locality lead to a lower contention so that the overhead of setting up and committing two transactions instead of one does not outweigh the additional computation required due the higher abort rate. The relative difference in run time is $-7.9\,\%$ compared with only one transaction. This example shows that even simple changes to an application yield a significantly different run time behavior and the return of investment of optimizations depends on many factors e.g., in-transaction times, contention, computation outside of a transaction, data locality. The fact that the EigenOpt approach determines and accounts for these parameters makes it valuable while the imprecise data that is collected due to the STM overheads and the intrusiveness of the approach must still be significantly improved.

6.5.6 Outlook for EigenOpt

The run time information required to extract and compute the necessary characteristics for EigenBench, put a high pressure on the event logging machinery and the frequent reading of hardware performance counters increases the intrusiveness. Our two examples show that the gains of the same optimization applied to two slightly different scenarios may

Testcase	Time in seconds	Abort rate (in $\%$)
Case1	7.21	23.54
Case1 with two transactions	2.72	4.32

Table 6.10: Test case with high contention and one and two transactions. Run with 8 threads on ExpX5670 and reporting averages over 100 runs.

Testcase	Time in seconds	Abort rate (in %)
Case1	9.03	2.19
Case1 with two transactions	9.74	0.79

Table 6.11: Test case with low contention. Run with 8 threads on ExpX5670 and reporting averages over 100 runs.

yield a speedup of s 2.65 x over the previous version of the code or a relative difference in execution time of $-7.9\,\%$ depending on the level of contention. We deduce from these findings that successful optimizations depend on many factors e.g., in-transaction times, contention, computation outside of transaction, data locality that all have to be taken into account. The EigenOpt approach considers all of these parameters which is the strong side. Unfortunately, the high intrusiveness biases the trace information so that the results presented for the mimicry of the `intruder` benchmark with EigenBench show a large deviation from the expected values. Hence, further research should investigate how to reduce the intrusiveness/probe effect on the TM application. A promising approach for TM applications using STM is sampling. Sampling takes statistical samples at a fixed interval and interpolates the application profile from these measurements. A sampling tool that also includes the reading of hardware performance counters is Open|SpeedShop. We believe that Open|SpeedShop could significantly reduce the intrusiveness while at the same time providing the required level of detail although with some uncertainty due to the statistical sampling.

6.6 Conclusions

In this chapter we present a framework and its components for the Visualization and Optimization of TM Applications (VisOTMA). VisOTMA supports the experienced as well as the untrained programmer of TM applications when designing, rating and optimizing a TM application. Through visualization of the TM application's behavior, we enable the programmer to identify pathological execution patterns that are known to degrade performance. Moreover, the question of a profitable transaction length is addressed in detail. In order to guide the programmer, we present a simple reference application and an optimization algorithm. Further, techniques to capture TM events and dynamic memory requests are employed. The resulting log files enable a comprehensive post-processing and visualization process even for unmanaged languages e.g., `C` or `C++`. Thus, our approach improves on the existing solutions through providing a correlation of TM events with the line in the source code and a mapping of addresses in transactional loads and stores to data structures of the application as well as collecting and providing the readings of hardware performance counters. Especially, combining the visualization (Paraver) with the comprehensive transactional statistics inside the VisOTMA framework is useful to uncover bottlenecks of TM applications.

We demonstrate the ability of the VisOTMA framework to identify the sources of conflicts in two well-known pathological TM cases (StarvingElder and FriendlyFire). We show how an inexperienced programmer may optimize the TM application even in the absence of performance-critical patterns. This is achieved by tuning the transaction size according to a metric provided by the VisOTMA framework. Through simply enlarging the transactions, an unexperienced programmer can tune a `C++` application simulating a fluid and yields a speedup up to 1.43 over the intuitive transactional version.

We highlight the versatility of the VisOTMA framework through visualizing the behavior of a hybrid TM system called TMbox. Optimizing the execution of `intruder` on TMbox illustrates that porting an application from STM to hybrid TM does not yield performance gains for free. Instead, a careful investigation of the run time behavior of the application combined with tuning the STM parameters yields the relative performance improvement of $24.1\,\%$ when moving from STM to a hybrid-ETL variant.

VisOTMA enables us to investigate the applicability of TM to the method of Conjugate Gradients. We carefully select a second formulation of the algorithm and use the hardware performance counters to compare the run time behavior of both algorithmic variants. We investigate the convergence behavior and dissect the utilization of the microarchitecture. With small extensions of VisOTMA components, we extent the capabilities to also trace the wait time at OpenMP barriers. This reveals the cause for the limited speedups with both variants.

Furthermore, we present TMPD, a component that enables the detection of execution phases in TM applications based on the conflict probability. Two algorithms for the detection of phases are transferred to TM: SignalAnalysis and a Wavelet-based scheme. We analyze most of the STAMP benchmarks with both algorithms and find that the threshold for detecting a phase change must be low and the window size must be small in order to detect phase changes. These parameters are not very encouraging when wanting to exploit these phase changes because it means that differences between phases are small and only exist for a short period of time. These findings make exploiting phase changes and achieving a speedup in a blocking STM system difficult.

Finally, we research how to simulate the effects of optimization with a configurable benchmark called EigenBench. The approach, named EigenOpt, relies on the readings of the hardware performance counters to extract the characteristic TM behavior of the TM application and obtain the parameter settings. In a simulation phase these settings are changed to reflect specific optimizations of the application. When this simulation yields performance gains, the optimization must be ported back to the application. Although this approach has been shown to work for simple test cases, we found that for more complex benchmarks, such as `intruder`, the intrusiveness of the tracing machinery must be reduced significantly in the future in order to achieve a higher precision of the parameters.

7. Compiler Support for TM and Guidance Through Static Information

The following Chapter 7 comprises two sections that address the compilation process of a TM application. Section 7.1 highlights the initial design and implementation of transactional memory support for the `C` programming language in GCC, as presented in [172, 174]. In addition to the major design decisions, further research directions and first results, this section also shows the progress of the GCC TM branch through evaluating later implementations and comparing the performed optimizations. Section 7.2 introduces an approach that uses static information to select a particular STM property, similar to [173]. Through a heuristic that analyzes the memory access patterns in transactions, the compiler decides between a word-based and a cache line-based conflict detection. An evaluation of this technique shows that it can be beneficial for novice programmers. Due to the special focus of both sections, each of the sections concludes independently of the other.

7.1 Towards TM for GCC

This chapter describes the design of a transactional extension for the C language, implemented in the GNU Compiler Collection (GCC). This design derives from the pioneering work of Intel [4]. Ali-Reza Adl-Tabatabai from Intel leads an important standardization effort with the goal to establish a common semantic (and memory model), syntax, Application Binary Interface (ABI), and interactions with existing programming languages and practices for Software Transactional Memory.

Participating in this important standardization effort is a necessary step towards a mature TM technology, upon which software developers and parallel computing research depend. In this context, we highlight some important ongoing research opportunities and challenges.

Transactional memory is a set of a parallel programming constructs and the accompanying programming patterns [91, 84]. It borrows database semantics, terminology and designs to address the atomic execution problem. In contrast to traditional low-level synchronization mechanisms, the programmer does not manage locks directly but relies on a more abstract,

```
1 int gvar;
2 int main () {
3    int a = 15;
4 #pragma tm atomic
5 {
6    gvar = ++a;
7 }
8    printf ("Global␣variable␣%d\n", gvar);
9 }
```

Listing 7.1: C extension with a pragma to specify transactions [172].

structured concept: an *atomic block*, hereafter called a *transaction*. From the programmer's point of view, atomicity is understood as two-way isolation of shared memory loads and stores within a transaction. From an implementation point of view, it allows for *optimistic concurrency*, with speculative conversion of the coarse-grain critical section into finer-grain, object-specific ones. The ability to correctly and efficiently transpose coarse-grain transactions into fine-grain, speculative concurrency is the key challenge for TM research and development. Both semantical and performance issues lead to a vast amount of studies and results [83]. Because of this implicit support for speculative execution, TM programming patterns generally include *failure atomicity* mechanisms, with programmer-controlled *abort* and *retry* constructs. These constructs are, for a part, complementary to parallel programming, and can improve the software development productivity at large.

Based on this design and implementation effort, we are conducting research on compiler optimizations to reduce the performance penalty of STM systems. We also study the potential of TM to support automatic parallelization, enhancing the support for generalized and sparse reductions in the automatic parallelization pass of GCC.

As related work has already been discussed, the design and implementation address the context of unmanaged languages only, with a *word-based instrumentation of shared memory accesses* in transactions. In this context, it is also natural to assume *weak isolation* of transactions with respect to non-transactional code; this comes with obvious limitations in terms of concurrency guarantees and cooperation with legacy code [83].

Many semantical variants of transactions have been proposed and investigated. The baseline semantics in our design is the one of a critical section guarded by a single lock, shared by all transactions. This choice is consistent with the original concept [91] and with most industrial designs; it offers composability and liveness guarantees, and is the only one for which a sound, intuitive and reasonably efficient weakly-consistent memory model has been proposed [133]. Our design is compatible with multiple transactional memory runtimes, facilitating its adoption in research environments and leveraging existing software support.

7.1.1 Design

This section presents the design decisions and additional mechanisms for TM support in GCC (called GTM). One of the major design goals is to be orthogonal to other parallel programming models. Thus, the implementation is not based on OpenMP.

We wish to support the optimistic execution of transactions, in the form of the simple example in Listing 7.1. To make this possible in C and in GCC, several enhancements are necessary. Besides some minor modifications to the C front-end to add support for the `#pragma tm atomic` and `__tm_abort`, we implemented two compilation passes: the *expansion* and the *checkpointing* pass. New GIMPLE tree codes `GTM_TXN`, `GTM_TXN_BODY`, and `GTM_RETURN` are introduced while parsing the transactified source code. The construction of the control flow graph is altered according to the OpenMP scheme for atomic sections: a basic block is split everytime a `GTM_DIRECTIVE` is encountered; this scheme simplifies the identification and management of transactions during the expansion pass.

7.1.2 Expansion

The first pass is called `gtm_exp`. It performs the following expansion tasks:

- function instrumentation, for all functions marked as callable from a transaction;
- recombination of the previously split basic blocks;
- instrumentation of shared-memory loads and stores with calls to the STM runtime — read or write *barriers*.

We currently instrument all pointer-based accesses. GCC's *escape* information will be used to later restrict this instrumentation to shared locations only.

In addition, the pass checks for language restrictions that apply for transactions. For instance, invocations of `__tm_abort` are only valid in the scope of a transaction. To access and process transactions conveniently, a `gtm_region` tree is built. The region tree facilitates the flattening of inner transactions.

7.1.3 Checkpointing

In case a transaction is rolled back, the effects on registers and stack variables have to be undone. The procedure to revert to the architectural state before entering the transaction consists of a call to `setjmp` combined with saving the contents of variables. We refer to this mechanism as *checkpointing*. An alternative to checkpointing variables, is to copy and restore the active stack frame as described in [61]. When the transaction rolls back the old stack frame is substituted for the new one to restore the previous state. Which of the two approaches is superior depends on the use case. If many variables are live-in to the transaction, copying a continuous amount of memory is expected to be faster than copying each variable exclusively. In case the amount of live-in variables is small compared to the active stack frame, copying and replacing variables is faster. We believe that the latter use case is more common. Thus, the second compiler pass implements the checkpointing scheme similar to the one in [202]. In addition the `setjmp`/`longjmp` mechanism is used to restore the actual register file. During the compiler pass one additional basic block is introduced. This basic block is connected via the control flow so that it is executed in case of a rollback and restores the values of variables. The saving of the values (and storing them into a temporary variable) is done before calling setjmp. In order to reduce the number of copy and restore operations, only variables that are live-in to the transaction are considered. The availability of liveness information require the pass to operate on SSA-form. For a seamless integration with the previous `gtm_exp` pass, the `gtm_checkpoint` pass removes the marker and adds the real checkpointing scheme. The outcome of this procedure

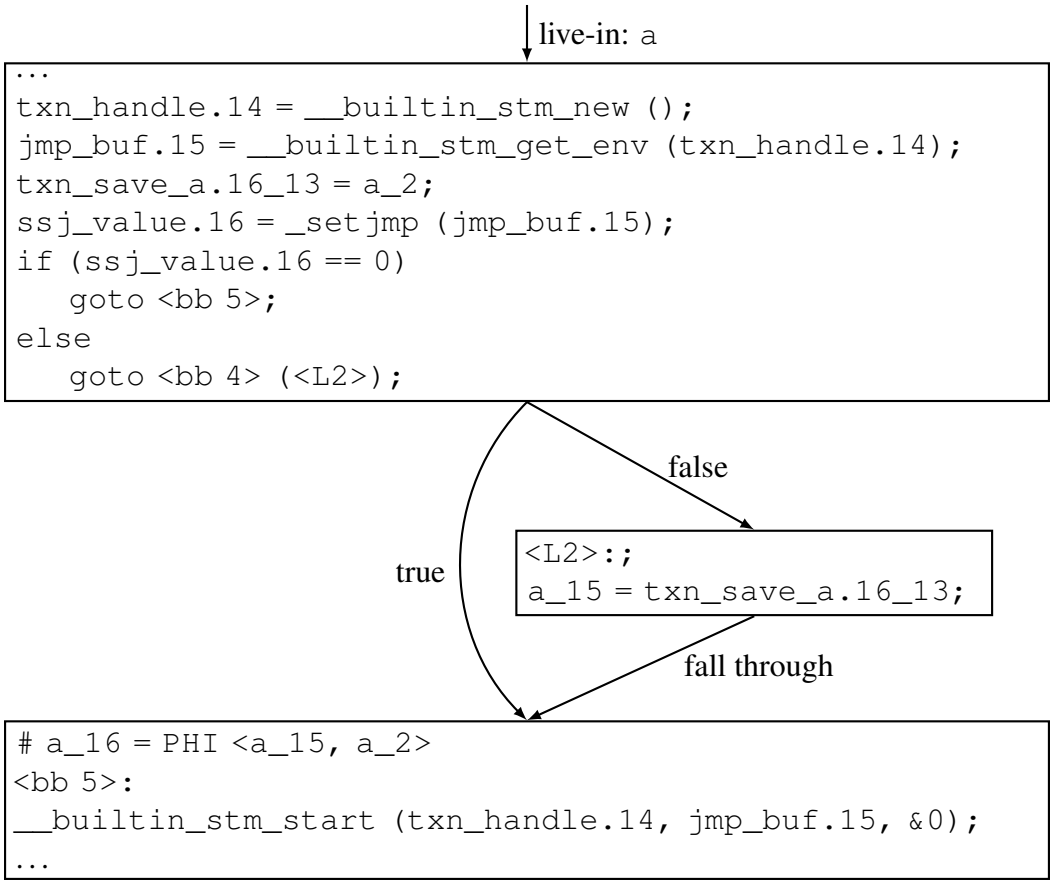


Figure 7.1: Checkpointing mechanism after the `gtm_checkpoint` pass.

is illustrated in Figure 7.1: the instruction sequence before the call to setjmp captures the value of the live-in variable `a` and saves it into the temporary variable `txn_save_a`. In case the transaction has to roll back, the library executes a call to `longjmp` and returns to the location where the `setjmp` was called. Thus, it returns from the setjmp with a non-zero return value. Subsequently, the basic block on the right hand side of Figure 7.1 gets executed and the value of the variable is restored to `a`. The Φ node on the next basic block merges the different versions of `a`.

7.1.4 Optimizations and Extensions

This paragraph outlines some opportunities and directions for optimization.

First exploiting the properties of the underlying intermediate representation (GIMPLE) yields some benefits. GIMPLE distinguishes between memory and register variables. Thus, a variable living in memory needs to be loaded into a register prior to being used. All memory loads are already assigned to a temporary variable. In order to reduce the number of introduced temporaries, the existing loads and stores could be directly substituted by calls to the STM run-time, reducing the number of temporary variables and, so, the work of optimizers.

Second the STM barriers, represented as builtins (or intrinsics), should make use of the function attributes provided within GCC. Optimizers determine the amount of valid optimizations depending on the function attribute. The current approach is to set an attribute signifying that the function call does not throw an exception for all barriers. Relaxing this

conservative choice for `stm_load` barriers to a *pure* attribute, usually used for functions not writing to memory, seems promising to enable few optimizations while preserving the correctness of the optimized code. Not all STM barriers qualify for relaxed attributes. For instance the `stm_start` and `stm_commit`-barriers enclosing the body of a transaction, are to remain as strict as possible. Otherwise store sinking or load hoisting optimizers may sink stores out of transactions and loads into them. Both optimizations potentially violate the intention of the programmer and weaken the boundaries set by transactions. Thus, the resulting code would not be correct.

The third optimization is to subdivide the passes in order to exploit the optimizations on SSA form. The *expansion* pass would be split into two phases. The remaining first part would only expand the `stm_start` and `stm_commit` barriers. Whereas the second part is placed at the end of the SSA optimization passes and introduces the `stm_load` and `stm_store` barriers. The proposed design utilizes the optimizations on SSA form and respects the properties of transactions.

When transactions occur in OpenMP parallel sections, we may rely on the shared/private clauses to refine the set of variables and locations to be instrumented by memory barriers. This optimization was proposed in previous transactional extensions to OpenMP [136, 9], but it may of course be designed as a best-effort enhancement of our language-independent TM design.

Further design and implementation of these optimization is under way in the context of the `transactional-memory` branch of GCC. This branch initiated from our design, and was opened in October 2008 by Richard Henderson (Red Hat). The branch targets the same ABI as Intel[1]. It implements an Inter-Procedural Analysis (IPA) pass to decide which functions to clone. GCC's exception handling machinery assures the correct treatment of transactions through the optimization passes. This ensures a conservative interaction with SSA-based optimizations although it does not prevent them entirely. The branch is fully functional by now and has been improved and merged with mainline GCC. With the release of GCC 4.7.0, transactional memory is available as an experimental feature.

Transactional environments require special mechanisms to enable developers to apply common programming patterns. It is the case of the *publication* and *privatization* patterns that frequently arise while programming with locks [133]: they feature concurrent accesses to shared variables inside *and outside* transactions. The absence of races is guaranteed by the lock semantics and by any weak memory model that subsumes Release Consistency (RC).

Semantical support for these patterns is particularly helpful when transactifying legacy code with non-speculative critical sections. Indeed, weak isolation and weak memory consistency models do not guarantee that such publication and privatization patterns will behave consistently with a lock-based implementation. Current STM designs propose *quiescence* as the mechanism to solve the problem occuring while one transaction tries to privatize a data member whereas the other tries to write into it [133]. Quiescence enforces an ordering of transactions so that transactions complete in the same order as they started. Besides allowing the programmer to use well known constructs and follow classical programming patterns, this mechanism comes with a significant performance penalty. We believe that further research in this area is inevitable to speed up the execution

[1] `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20#ABI`

of transactions while retaining a consistency model compatible with easy transactification of lock-based code — here *single-lock semantics* is sufficient [133].

Calling legacy code from inside transactions constitutes another problem for programming in a transactional environment, because effects of these functions can not be rolled back. The same holds true for system calls. The solution is to let transactions execute in, or transition to, *irrevocable mode*. The runtime assures that the irrevocable transaction is the only one executing and, thus, can not conflict with other transactions. Hence, the transaction runs to completion. [206] presents possible implementations and applications of irrevocable transactions, whereas [195] also evaluates different optimized strategies to implement irrevocability. Further research concerning irrevocability could benefit from the presented implementation.

Link-Time Optimization (LTO) as well as Just-In-Time (JIT) compilation are well-known compilation approaches that are not yet extensively applied to transactional workloads. The former has a high potential for pointer-analysis-based optimizations (like escape analyses to eliminate unnecessary barriers), while the latter can substitute dynamic code generation and transaction instrumentation rather than static cloning of functions callable from transactions.

7.1.5 Parallelization of Irregular Reductions

Reduction operations are a computational structure frequently found in the core of many irregular numerical applications. A reduction is defined from associative and commutative operators acting on simple variables (scalar reductions) or array elements inside a loop (histogram reductions). If there are no other dependencies but those caused by reductions, the loop can be transformed to be executed fully parallel, since — due to the associativity and commutativity of their operands — iterations of a reduction loop can be reordered without affecting the correctness of the final result.

Currently, the automatic loop parallelization pass in GCC is capable of recognizing scalar reductions. Once the reduction pattern has been detected the code generation step relies on the OpenMP expansion machinery to distribute iterations of the loop into several threads. Reduction parallelization employs a privatization algorithm: the transformed loop has a parallel prefix, where each thread accumulates partial results in local copies of the reduction variable, followed by a cross-thread merging phase in which partial results are combined into the shared (reduction) variable.

The reduction recognition routine in the automatic parallelization pass can be extended to detect *sparse* reductions. Sparse reductions correspond to inductive dependences on the reduction variable/array that only exists for a fraction of the loop iterations. This is generally the case for moderate-to-large reductions with indirection variables (e.g., histograms), as well as some reductions guarded with data-dependent control flow.

A parallel reduction algorithm has to be chosen for the code generation of sparse reductions. A simple solution is that of enclosing the accesses to the reduction variable/array within a critical section. The main drawback of the typical lock-based implementation of this method is that it exhibits a very high synchronization overhead. We can leverage the infrastructure for transactional memory programming within GCC to devise an alternative approach to reduction parallelization in which we enclose the critical section in an atomic transaction, and let the underlying STM runtime detect and resolve possible conflicting

```
1  int image[1024][768];
2  int main ()
3  {
4    int hist[256];
5
6  #pragma omp parallel for
7    for (i=0, i<1024; i++)
8      for (j=0; j<768; j++)
9      {
10       /* Some reduction-independent
11        * work parameterized with M
12        */
13       WORK;
14       pixval = image[i][j];
15       /* Begin critical section */
16       hist[pixval]++;
17       /* End critical section */
18     }
19 }
```

Listing 7.2: Example of a reduction pattern with nested loops [172].

accesses to same array locations. Many scientific and numerical applications operate on large and sparse datasets; they are amenable to transactional parallelization since we can optimistically assume that only few conflicting accesses to the same memory locations will manifest at runtime.

As a motivating example we show here the results of parallelization of a synthetic loop containing a histogram reduction, see Listing 7.2. To model the effect of varying amount of work besides the reduction within the loop, we employ a WORK section consisting in an additional loop nest which iterations have been parameterized with variable M. This loop only operates on data independent of the reduction operation. Loop iterations are distributed between 4 threads, and we compare three different synchronization schemes, namely locks with Pthreads, OpenMP `critical` directive and transactions. The latter is achieved through the use of the `#pragma tm atomic`. Calls to the STM library (TinySTM v.0.9.0b1 [62]) for read/write barrier instrumentation and transaction rollback/restart are automatically instantiated by the GTM compiler.

To account for the effect of different degrees of contention, we consider histogram creation for two synthetic images: a completely black image (our worst case), and an image with randomly generated pixel values.

We show in Figure 7.2 the results of the execution of the example loop on a Intel Core 2 Quad CPU (L2 cache with 4MBytes). On the x-axis we consider increasing amounts of work in the loop body by increasing the value of the parameter **M**. On the y-axis we plot the speedup of the various parallelization schemes against the serial version of the loop.

In the random image there is low contention for array locations, and the performance of the optimistic TM approach is always better than the others. We can achieve speedups with this technique even for small values of M. However, as expected, high contention on array

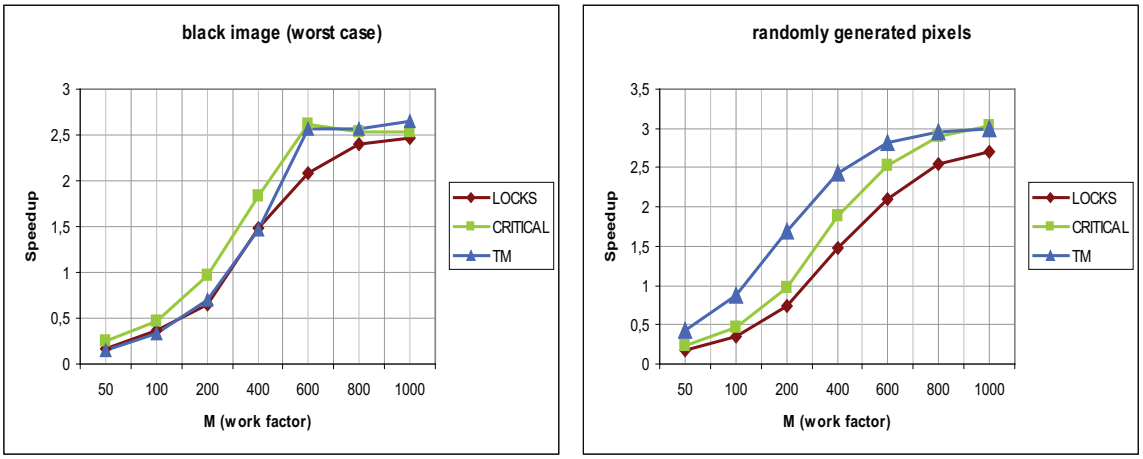


Figure 7.2: Speedup over sequential execution using locks, OpenMP critical sections, and transactions; first published in [172].

elements has a strong impact on the performance of TM. This can be seen comparing the trend of the TM curve in the two plots, and is justified by the fact that the overhead for frequent transaction rollback and restart is greater than that carried by the locks. Clearly this behavior is also affected by the value of M. When there is little work besides the reduction in the loop the overhead is predominant for all of the proposed techniques, but in high-contention scenarios TM is the one that is mostly affected by this parameter, not only because it influences the frequency of aborts and rollbacks, but also because the overhead for starting and committing a transaction is not amortized by other computation.

These two factors are directly related to the sparsity of the dataset on which we operate and to the granularity of the transaction, for this reason we consider them as the two main parameters to be investigated in real workloads to discover the boundaries wherein a reduction parallelization algorithm that exploits transactions is successful. First experimental results are encouraging, since they show that adequately tuning these parameters, a transactional approach to irregular reduction parallelization can bring significant performance improvements with respect to the use of locks.

7.1.6 Overinstrumentation with GCC

Another experiment aims at quantifying the influence of over-instrumentation on the performance of a TM application. For this experiment, we select `kmeans` from the STAMP benchmarks suite and compare the run times of an hand-instrumented and a GTM-instrumented version. kmeans1 stands for execution with the low contention parameter set whereas kmeans2 represents the high contention case. This section presents a comparison of compiler translated and hand instrumented atomic blocks. As compiler we partially ported the transactional memory enabled GCC, which is available under `http://gcc.gnu.org/svn/gcc/branches/transactional-memory/`, to generate code for TinySTM. The *trans-mem* branch originates from the previously presented GTM design, but was revised and enhanced by Richard Henderson (Red Hat). The revision addressed the integration of transactions with GCC's exception handling infrastructure and the integration with the interprocedural analysis passes and optimizers.

The experiments are conducted on our experimental platform ExpE5410. The operating system is a x86_64 GNU/Linux based on a kernel version 2.6.24-22-xen and virtualization

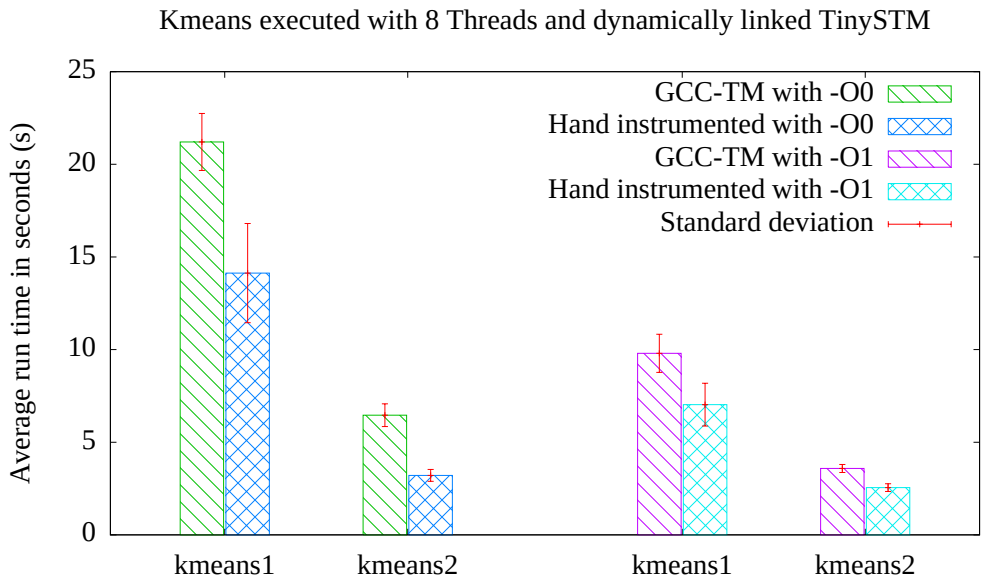


Figure 7.3: Run time of benchmark `kmeans` with compiler and hand instrumented transactions.

extensions. The STM library is TinySTM in version 0.9.9. The benchmark `kmeans` is taken from the STAMP benchmark suite (version 0.9.10) and is run with two input data sets [24]. The first one has low contention whereas the second one suffers from high contention - both are run with the largest input data sets. Run times of nine runs are aggregated and averaged to generate reliable results. Figure 7.3 presents the average run times together with the standard deviation. Different compiler optimization levels are shown: `-O0` no optimization and `-O1` with a basic set of optimizations. The results for `kmeans` show that optimized compiler version performs better than the unoptimized one, although still yielding a slow down of $\approx 40\,\%$ compared to the hand instrumented code.

A closer examination of the various assembly files of `kmeans` reveals that both hand instrumented versions contain 4 read and 4 write barriers. The compiler instrumented transactions contain 16 read and 4 write barriers without optimizations and 8 read and 4 write barriers with basic optimizations enabled. These additional barriers explain the large differences in run time and beg for more advanced optimizations to identify and eliminate these unnecessary barriers. In the following section, we continue this experiment to quantify the impact of advanced optimizations on the performance.

Yoo et al. report false sharing with kmeans that accounts for $40\,\%$ of the aborts [214]. Surprisingly the benchmark shows a good scalability despite the false sharing. In their experiments, the compiler instrumented the transactions whereas in ours the programmer or the compiler did the instrumentation. As seen in the paragraph above, kmeans does benefit from the programmer's application-level knowledge that avoids to instrument pointer dereferencing. Since Yoo et al. also apply further optimizations that pass these knowledge to the comiler, we would have expected a greater gain. More specifically they introduce the `#pragma tm_waiver` to signal to the compiler that the corresponding block or function does not need instrumentation. Given our previous findings on the barrier instrumentation

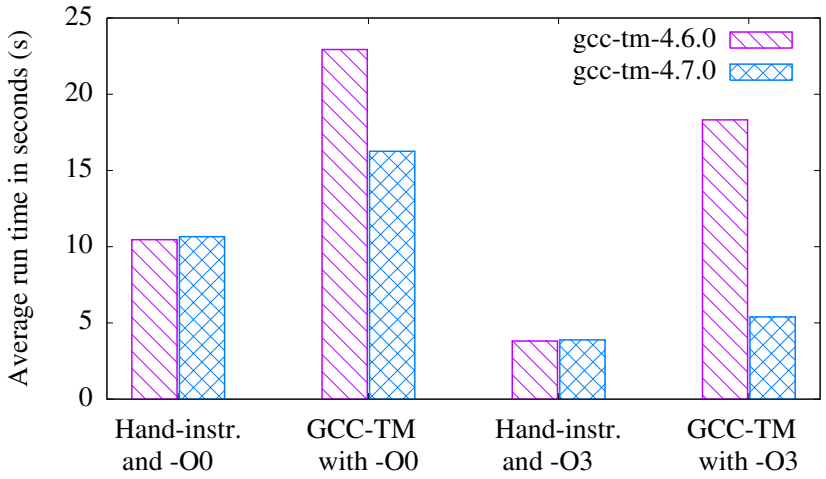


Figure 7.4: Comparison of the run times of Kmeans with low contention and hand-instrumented or compiler-instrumented transactions compiled with gcc-tm-4.6 or gcc-tm-4.7.

of Kmeans with GTM, we would expect these annotations to reduce the number of barriers for the pointer dereferencing and yield a speedup. The reported differences in run time in their results are small.

7.1.7 Improvements with GCC-4.7

In this section we study the impact of advancing the TM-specific compiler optimization in GCC and implementing the Application Binary Interface (ABI) for STMs. The ABI is the product of a standardization process that was initiated by Intel. The goal is a uniform application binary interface for STMs that is widely supported through industrial and academic STM systems. The ABI decouples the compiler from the STM system and ensures the interoperability of compilers and STMs that have been developed independently. The transactional memory branch targets this ABI. Richard Henderson from Red Hat Inc. maintains the transactional memory branch and contributed important parts of the implementation as did Aldy Hernandez.

Our goal is to quantify the effects of the advanced TM-specific compiler optimizations on the performance of the application. In order to achieve this goal, we use an early checkout from the transactional memory branch (version 4.6.0 showing the 13^{th} of April 2010 as date) and compare it with the final merge of the transactional memory branch into mainline GCC. Here we use the version 4.7.0 that has been released on the 22^{nd} of March 2012. Both versions come with the corresponding libITM that also evolved over time. We are going to refer to these versions as *gcc-tm-4.6* and *gcc-tm-4.7* respectively. Of course advancing the compiler from revision number 4.6 to 4.7 comes with many more changes than just TM-specific optimizations[2]. Thus, we need to find a way to distinguish the impact of newly introduced TM optimizations from general optimizations that also have a benefit for TM. Therefore we use the hand instrumented kmeans application from the STAMP benchmark suite as a baseline. As STM we are again using TinySTM (version

[2] `http://gcc.gnu.org/gcc-4.7/changes.html`

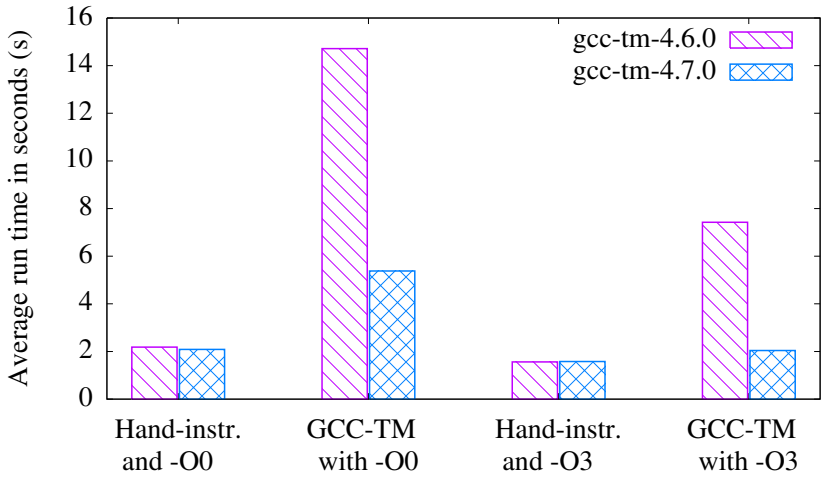


Figure 7.5: Comparison of the run times of `kmeans` with high contention with gcc-tm-4.6 and gcc-tm-4.7 and hand-instrumented and compiler-instrumented transactions.

0.9.9) which is also compiled with the compiler under test and a fixed optimization level of -O3. This mimics the shipping of an optimized STM to the customer. The application is compiled with the compiler under test setting the reported optimization level. All reported results are averages over 30 runs with 8 threads on the experimental setup ExpX5670 (cf. to Section 4.3).

Figure 7.4 illustrates the run times of Kmeans with low contention: for hand-instrumented transactions addressing TinySTM compiled with gcc-tm-4.6 or gcc-4.7 (without setting the flag to trigger any TM passes) with an optimization level of -O0 the run time is ≈ 10 s. When GCC does the TM instrumentation (indicated with *GCC-TM* in the figure), there is a huge gain from gcc-tm-4.6 (run time ≈ 23 s) to gcc-tm-4.7 (run time ≈ 16 s). We observe a similar trend with optimization level -O3: while the run time with hand-instrumented transactions even increases from ≈ 3.8 s to ≈ 3.9 s, the GCC-instrumented transactions decrease from ≈ 18.3 s to ≈ 5.4 s when going from gcc-tm-4.6 to gcc-tm-4.7.

Similarly Figure 7.5 demonstrates the impact of advancing the GCC compiler on Kmeans with high contention. Again, hand-instrumented transactions that target TinySTM show only very small improvements. With optimization level -O0 the run time improves from ≈ 2.2 s to ≈ 2.1 s and stays constant with optimization level -O3 with ≈ 1.6 s. For the instrumentation with GCC-TM the findings are different: with -O0 the run time improves from ≈ 14.7 s to ≈ 5.4 s and with -O3 from ≈ 7.4 s to ≈ 2.0 s.

In order to make the previous measurements straightforward to comprehend, Figure 7.6 shows the speedup of gcc-tm-4.7 over gcc-tm-4.6. The speedup is computed according to the equation $\frac{\text{Run time}_{gcc-tm-4.6}}{\text{Run time}_{gcc-tm-4.7}}$. The hand-instrumented cases with TinySTM do only marginally benefit from the compiler improvement (or may even suffer a small degradation as with -O3). On the other hand the GCC instrumented transactions show speedups between ≈ 1.4 for kmeans with low contention and optimization level -O0 and ≈ 3.6 for kmeans with high contention and optimization level -O3. These performance numbers show that TM-specific optimizations have a far larger impact than other improvements.

These huge differences in run time need to be invested closer to identify the dominant

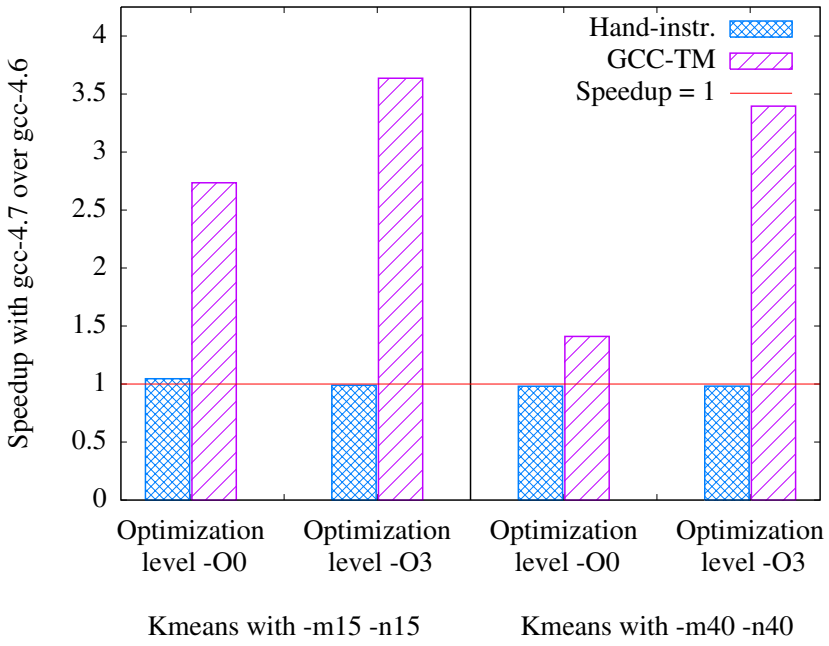


Figure 7.6: Speedup of gcc-tm-4.7 over gcc-tm-4.6 measured with `kmeans` with high (left hand side) and low contention (right hand side) to study the impact of optimizations on hand-instrumented and compiler-instrumented transactions.

optimizations. These differences in the implementation of the TinySTM and the libITM requires a precise investigation of the number and type of read and write barriers issued to clearly separate compiler improvements from improvements of the run time system. Similar to the experiments in the previous Section 7.1.6, we are going to investigate the number and types of barriers that GCC issues for kmeans.

Table 7.1 holds the read barriers issued with the optimization levels O0 and O3 by gcc-tm-4.6 and gcc-tm-4.7. With -O0 specified both compilers generate only basic full barriers. In the table these basic barriers (e.g., RU4 stands for read unsigned 4 Bytes) form the first group. Barriers in this group perform the intended task but do not exploit the static knowledge of the compiler. Barriers in the second group (e.g., RfWU4 named read-for-

Types of read barriers	gcc-tm-4.6 with -O0	gcc-tm-4.6 with -O3	gcc-tm-4.7 with -O0	gcc-tm-4.7 with -O3
RU4	2	0	2	0
RU8	9	2	9	2
RF	5	2	5	1
RM128	0	1	0	2
RfWU4	0	1	0	1
RfWU8	0	1	0	1
RfWF	0	1	0	2
RfWM128	0	1	0	0
Total	16	9	16	9

Table 7.1: gcc-tm-4.6 and gcc-tm-4.7 with two optimization levels generate read barriers.

Types of write barriers	gcc-tm-4.6		gcc-tm-4.7	
	with -O0	with -O3	with -O0	with -O3
WU4	1	0	1	0
WU8	1	0	1	0
WF	2	1	2	0
WM128	0	0	0	1
WaWU4	0	1	0	1
WaWU8	0	1	0	1
WaWF	0	1	0	2
WaWM128	0	1	0	0
Total	4	5	4	5

Table 7.2: Comparison of the write barriers issued by gcc-tm-4.6 and gcc-tm-4.7 with two optimization levels.

write unsigned 4 bytes) are generated by the compiler when the transaction also writes the address it is currently going to read. With this special barrier (and with an STM that supports this kind of optimization), the STM omits accessing the data first as a reader and later as a writer. Instead it directly accesses the data as a writer (without actually writing to it) which avoids the need to upgrade from a reader to writer lock internally in the STM. The later write access only has to provide the new value. With optimization level O3, two findings holds for both versions of the compiler: the total number of read barriers decreases from 16 to 9 and the number of barriers from the second group rises from 0 to 4. A closer look reveals that gcc-tm-4.6 selects 2 RF and 1 RM128 while gcc-tm-4.7 selects 1 RF and 2 RM128 barriers. gcc-tm-4.6 chooses 1 RfWF and 1 RfWM128 barrier whereas gcc-tm-4.7 issues 2 RfWF barriers and no RfWM128 barrier.

Table 7.2 illustrates the write barriers generated by both compilers and two optimization levels. With optimization level -O0 both compilers only select basic barriers from the first group. With -O3 enabled, gcc-tm-4.6 and gcc-tm-4.7 mainly select barriers from the second group. Only one basic barrier remains. With gcc-4.6 this barrier is WF and with gcc-tm-4.7 WM128. Another difference is that gcc-tm-4.6 issues 1 WaWF and 1 WaWM128 barrier whereas gcc-tm-4.7 selects 2 WaWF and no WaWM128 barrier.

The base optimization case with -O0 is the ideal case to quantify the impact of the improvement of the libITM run time on application performance because gcc-tm-4.6 and gcc-4.7 generate the same amount of barriers of the same type. Thus, the performance of libITM run time system dominates the performance of the TM application. The experiments with hand-instrumented transactions and TinySTM that showed only a marginal improvement (if any) support this claim. Figure 7.6 clarifies that for -O0 the gcc-tm-4.7 compiler and libITM run time system yields a speedup from ≈ 1.45 up to ≈ 2.95 depending on the contention for kmeans. Thus, we ascribe these speedups to an optimized implementation of the libITM run time system. With -O3 the gains of gcc-tm-4.7 are even greater: the improved selection of the optimized read-for-write and write-after-write barriers, yield a speedup of ≈ 3.57 over gcc-tm-4.6. These results demonstrate that the gcc transactional memory branch, that is maintained by Red Hat Inc., makes progress and alleviates the overheads of STM through a combined compiler and run time approach. On the other hand, these barrier counts also show that there is still a long way ahead because the number of compiler generated barriers still differs from the number of necessary barriers that is achieved with hand-instrumentation (4 read and 4 write barriers). In particular the derefer-

encing of pointers to locations that have not escaped yet can be avoided in the considered case. Due to the conservative assumptions of the compiler, removing these barriers is not feasible as of now. A future more general analysis could focus on link-time optimizations. At link time, potentially separately compiled modules/external libraries are linked together resolving the defined symbols. Thus, the linker has a broader view on the definition and the usage of symbols than the compiler. With this knowledge a whole-program analysis combined with an advanced alias-analysis may safely omit barriers because the linker is able to exclude the external linkage of a symbol.

7.1.8 Concluding Remarks for TM in GCC

We presented an initial transactional memory extension of the GNU Compiler Collection, and stressed its language-independent and STM-oriented design (yet compatible with hybrid hardware/software implementations). This integration also provides the foundation to integrate TM in an enhanced automatic parallelization strategy, where much of its design and implementation can be reused for the parallelization of sparse, generalized reductions. For these we demonstrate possible performance gains with a manual version. We also highlighted key optimization challenges and opportunities; together with Yoo et al. [214] and the more pessimistic study of Cascaval et al. [27], we stress the importance of compiler and joint language-compiler studies for the future adoption of TM in real world applications. To assess the quality of the ongoing work on the transactional memory branch (that originates from the presented GTM design) by Red Hat Inc., we study the evolution of the compiler optimizations and their impact on the performance of the TM application. We conclude that GCC's compiler and the libITM run time system already made very good progress in tackling the overheads associated with STM and identified directions for future optimizations.

7.2 Selection of the Conflict Detection Granularity in an STM

Many researchers advanced the state-of-the art in TM run time systems, as we already mentioned in Chapter 2. A favoured subject is the conflict detection algorithm that can differ in *when* conflicts are detected (so called early or late detection) and *where* speculative data is buffered (in a dedicated buffer or in-place). Another important design aspect addresses the granularity with which conflicts are detected. Some designs feature conflict detection on word size whereas others favor conflict detection on cache line size. In the absence of hardware transactional memory, programmers, who are interested in a new synchronization paradigm, usually utilize a software-based solution, called Software Transactional Memory (STM). Some STMs are highly parameterizable forcing the programmer to choose the desired STM properties. However, different applications prefer different STM characteristics. Since the programmer may not be aware of the preferred STM system for the application, we aim to guide her through compiler support. This approach relies on static information of the program to select a particular STM property. The compiler, more precisely the Low Level Virtual Machine framework [119], is extended with an analysis pass that analyzes memory access patterns in transactions [59]. The findings are then processed by a novel algorithm which proposes a specific STM system. Thus, based on the analysis of the source code, the programmer may select a fitting STM system.

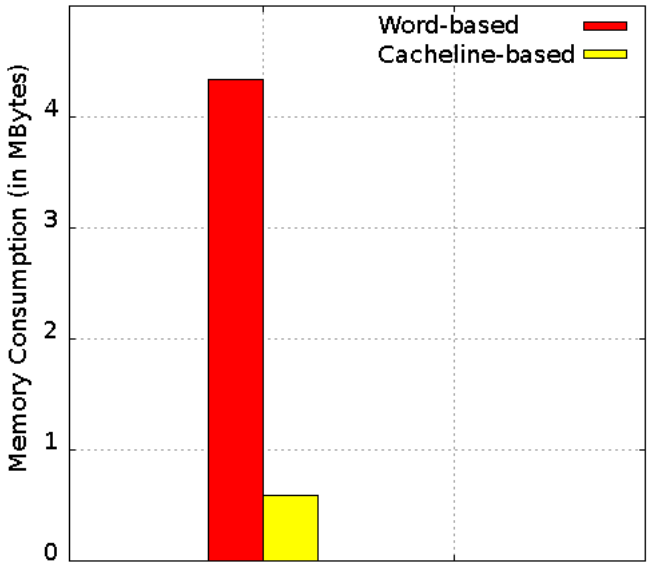


Figure 7.7: Comparing the memory consumption of a word-based (left bar) and a cache line-based (right bar) conflict detection scheme (similar to [173, 59]).

The focus of this work is on the conflict detection granularity in STM systems. We seek to distinguish between word size and cache line-based conflict detection. One of the issues with cache line-based conflict detection is the occurrence of false sharing. False sharing occurs when different parts of the same cache line are accessed by different processors. In terms of application behavior there is no sharing of data, but due to the cache coherence protocol that operates at the granularity of a cache line, the false sharing is classified as true sharing [178]. An invalidate-based coherence protocol will invalidate the cache line of the distant processors upon a write. On the distant processors the next access to that cache line will result in a miss. The overhead due to the repeated transfer of the cache line results in a loss of performance.

In the context of a multi-processor system, false sharing results in cache miss and stall times [86], whereas in the transactional memory domain, false sharing results in falsely detected conflicts. These false positives result from a false sharing of STM internal locks and are resolved by aborting one of the involved transactions. Consequently, all computed values are rolled back and execution resumes from the beginning of the transaction. Thus, one false conflict may render all previous computations in a transaction useless.

For this research we use a state-of-the-art software transactional memory system called TinySTM [62]. TinySTM uses a time-based TM algorithm to ensure the transactional properties. Conflicts are originally tracked and detected on a word granularity. A proper conflict detection is assured by employing locks that are held in a lock array. In case a transaction loads data from or stores data to a memory address, the address is fed into a hash function that maps it to a lock in the array. In a word-based conflict detection scheme, each word is mapped to a lock. For a cache line-based conflict detection all addresses in the same cache line (that we determine to be 64 Bytes) hash to one lock. This coarsens the granularity at which conflicts are detected and reduces the amount of locks required to cover the same memory space. Figure 7.7 illustrates the different memory requirements for both variants. The memory consumption is measured with `valgrind` and reveals that the lock array dominates the memory consumption. Thus, the cache line scheme consumes significantly less memory than the word-based scheme. However,

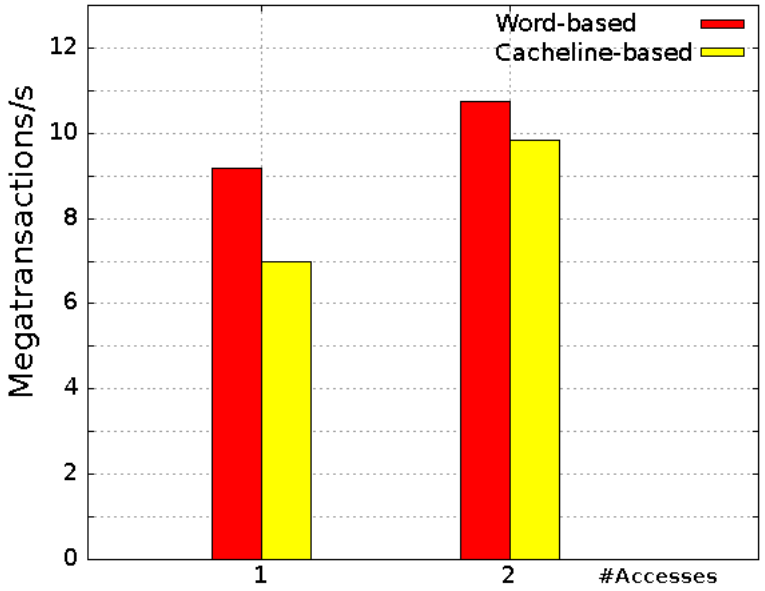


Figure 7.8: False sharing in the context of transactional memory: the throughput of a word-based (left bar) and a cache line-based (right bar) conflict detection scheme are compared (taken from [173, 59]).

this advantage in memory consumption comes at the prize of being vulnerable to false sharing. Figure 7.8 compares the throughput of a word-based and a cache line-based conflict detection granularity. In this use case two threads exercise a false sharing pattern. The amount of adjacent accesses of one thread to the same cache line varies between 1 and 2. Both scenarios exhibit false sharing but differ in throughput. This example clearly demonstrates the downside of a cache line granularity as in both cases the word scheme has a higher throughput. Thus, an application exhibiting a similar memory access pattern that is run with cache line granularity would have suffered a slowdown.

This example demonstrates that a decision between cache line and word granularity influences two aspects of the run time behavior of the application. First, the memory requirements, since cache line granularity demands less memory. Second, the run time as a false sharing pattern leads to an increased amount of rollbacks (only cache line granularity). Since this decision should not be taken lightly, we present a compiler-based approach to decide, which alternative promises better performance.

7.2.1 Detection of Memory Access Patterns in Transactions

In this section, the solution for the detection of memory access patterns in transactions (short MAPT) is presented. The MAPT is logically divided into three phases. First, the data aggregation phase gathers relevant information about transactional memory accesses. Second, an analysis phase processes and analyzes these data. Third, a decision phase ranks the findings and decides in favor of one STM system. Figure 7.9 illustrates the structure of this approach.

The implementation of MAPT is embedded in the Low Level Virtual Machine (LLVM) framework. LLVM does not natively support the detection of parallel functions. Thus, we present a method to identify parallel functions first. In this approach, functions, that are identified for parallel execution, are assumed to run in parallel. Inside these functions only transactional memory accesses are of importance to the analysis. Subsequently, a

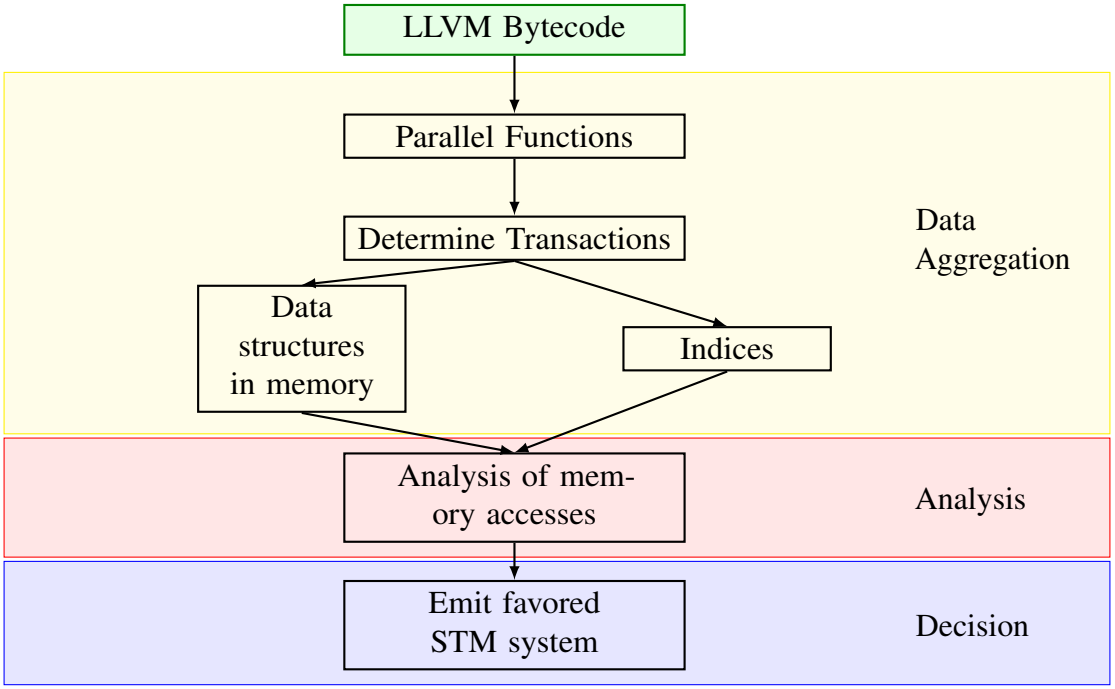


Figure 7.9: The LLVM compiler pass is divided in three logical phases: data aggregation, analysis and deciding which STM to select (cf. to [173, 59]).

pass collects information on transactions and data structures accessed inside transactions. The following analysis detects false sharing. Therefore the pass records all indices for transactional accesses. These data constitutes the basis for the analysis of memory accesses. The analysis decides in favor of one STM system as a result.

Data Aggregation

The following paragraphs provide details on the implementation and integration of MAPT in the Low Level Virtual Machine framework.

Parallel Functions are not identified natively in the Low Level Virtual Machine framework. This is due to the fact that many languages (e.g., C) were developed for sequential execution. Thus, library-based approaches (e.g., Pthreads) add the concept of threads. A programmer utilizes an API to create, destroy and synchronize threads. In the MAPT pass the knowledge about this API serves as an entry point to identify functions which may run in parallel.

In particular MAPT targets the POSIX thread model Pthreads. In that case, a programmer creates a new thread by calling `pthread_create`. Thus, a first step in the MAPT pass is to find the call and extract the arguments. Thus, the LLVM intermediate representation is scanned for a call instruction with that particular function name. Then, the third argument of this function call is processed. This argument contains a pointer to the function which will be executed by the newly spawned thread. In search for not yet processed functions called from this one, MAPT recursively visits functions with a static call site. The use of function pointers combined with a dynamic call site leads to functions escaping this analysis. This issue needs to be addressed in the future in order complete the MAPT approach. Further, there is the risk of entering an endless loop when processing recursive function calls. We avoid this by storing all processed functions in a table and checking the table before descending.

Transactions are identified by examining the parallel functions. Calls to the STM run time system define these transactions. In particular calls to `stm_start` and `stm_commit` embrace a transaction body. The body contains transactional loads and stores from and to memory locations.

Data Structures. An `stm_load` takes the address of the memory location as argument. Relating the address and the corresponding data structure requires some work. LLVM represents a load-store architecture which means that each address is loaded into a register first. Thus, MAPT tracks the register content back to the name of the data structure. Often, this involves also tracking the index in a static data structure (e.g., array accesses). In case the index belongs to a simple loop which can be analyzed by the LLVM loop analysis, the information about loop bounds and iteration width is also of interest. Then, our pass records the gathered data in a table. These data is grouped according to the identifier of the thread and contains the name of the data structure and the possible values of the index (in case this information is available). In order for this approach to work, the static analysis needs a hint with how many threads the program is going to be executed. Thus, the programmer has to provide these information to the MAPT pass by setting a variable in LLVM.

Further, the current scheme only covers simple (1-dimensional) loops and does not use pointer analysis to detect aliasing pointer accesses to data structures. More refined pointer analysis techniques and advanced loop structures must be part of future research.

Analysis Phase

The analysis phase processes the global data gathered in previous steps. The algorithm for the analysis is split in two parts. First, false sharing is detected and identified. Second, a heuristic is employed to cover access patterns which result in accesses to the same elements in one cache line.

Detecting False Sharing is a major concern of this approach. These cases are identified by comparing the index sets of different threads. The width of a memory access is determined through the data structure it accesses. Therefore, computing the amount of elements per cache line is feasible. By computing the intersection of the index sets for different threads, a false sharing pattern can be detected if the following two conditions are met: (a) the data structure is aligned at a cache line boundary in memory; (b) the computed intersection is empty but indices from both sets fall within one cache line. The size of the cache line can not be derived statically and is provided by the programmer. The microarchitecture defines the size of the cache line. Therefore, new processors may require different settings. For our use case, the cache line size is set to 64 Bytes.

Heuristic for accesses to the same cache line. This heuristic complements the detection of performance-critical memory access patterns (e.g., false sharing) by enriching the decision process with empirical data. Based on profiling one example application (that performs concurrent memory accesses to the same cache line), a metric (also called heuristic) is extracted. A-priori it is not known which conflict detection granularity enables a higher throughput in this use case. This heuristic helps to solve the issue and decides for one conflict detection granularity. This allows to decide on a broader basis if more performance-critical memory access patterns are not detected. Table 7.3 illustrates the turnover point in the throughput (in $\frac{Txns}{\mathrm{s}}$) of the two TM conflict detection granularities when the number

Accesses per cache line	Throughput $[10^5 \frac{Txns}{s}]$ Word-based (tp_W)	Throughput $[10^5 \frac{Txns}{s}]$ CL-based (tp_C)	Relative Difference $\frac{tp\_C - tp\_W}{tp\_W}$
16	4.7	5.6	19.1%
8	9.0	9.7	7.8%
4	15.0	16.0	6.7%
3	19.9	20.3	2.0%
2	24.1	23.2	−3.8%
1	37.1	35.3	−4.9%

Table 7.3: Throughput of both conflict detection variants profiled with an example application with overlapping memory accesses (similar to [173, 59]).

of consecutive accesses to the same cache line varies. Additionally, the influence of a second parameter that defines the number of elements in the shared data structure, has been integrated (without Figure). The relative difference ($\frac{(tp\_C - tp\_W)}{tp\_W} * 100$) ranges from -4.9% to 19.1% (cf. Table 7.3). In the first case the word-based solution provides higher throughput, whereas in the latter case the cache line-based scheme is superior. Based on the measured values, a turn-over point is calculated such that this heuristic can be integrated in the algorithm. Since the heuristic is based on evidence from real executions (and thus influenced by the memory subsystem etc.), the previous profiling run of the example application have to be repeated when targeting a new architecture and a new turn-over point must be identified.

Decision

For now, we employ a simple decision scheme: in case a performance-critical memory access pattern (e.g., false sharing) is detected, the decision is in favor of a word-based granularity. In case accesses to the same cache line are detected, the heuristic decides which of the two granularities to use.

7.2.2 Evaluation

Experimental Setup

The experimental setup comprises an AMD Athlon(TM) 64 X2 Dual Core Processor running at 2 GHz with 4096 MBytes of main memory. TinySTM (in version 0.9.9) has been adjusted to obtain two different versions: one using conflict detection on a word, one on a cache line granularity. The LLVM framework (version 2.5) has been enhanced with capabilities for analysis of memory access pattern in transactions (MAPT). The enhanced framework is evaluated using two manually written test cases and one bank application to demonstrate the validity of this approach. These tests are complemented by two benchmarks from the STAMP transactional memory benchmark suite to emphasize performance gains on real world transactional memory workloads [24]. Transactions are instrumented manually with calls to the TinySTM. However, the influence of hand-instrumentation on the MAPT pass should be low. For some applications the number of STM calls in transactions differs from the compiler-instrumented version [174].

Test Cases

First, results of evaluating three test cases are presented in Table 7.4. Performance changes are reported as relative improvements according to $\frac{(tp\_sel - tp\_non)}{tp\_non} * 100$. The throughput of

Test case	Word-based	Cache Line-based	MAPT proposes	Difference (in %)
separate	9 173 365	6 991 694	Word Granularity	31.2%
overlap	473 177	540 424	CL Granularity	14.2%
bank	396 824	390 573	CL Granularity	−1.6%

Table 7.4: Throughput in transactions per second of three test cases (cf. to [173]).

Benchmark	**K-means**	**Bayes**
Word-based	6.1 s	7.1 s
Cache Line-based	5.2 s	5.9 s
MAPT proposes	Cache Line	Cache Line
Difference	14.7%	16.9%

Table 7.5: Run times of selected STAMP benchmarks (as in [173]).

the selected variant (named *tp_sel*) is compared with the variant which was not selected (named *tp_non*). A higher throughput is better and, thus, an improvement. In the first test case, dubbed *separate*, different threads access separate elements in the same cache line. The MAPT analysis proposes the word granularity which results in an improved throughput. Second, *overlap* showcases how overlapping accesses are treated. The compiler favors cache line granularity which results in improved throughput. For *bank*, a TM application managing accounts in a bank, the cache line granularity is proposed but results in a small slowdown compared with word size. Inside the bank implementation the accounts for transfers are chosen at random. This makes a static analysis, for this part of the application, infeasible. As a result MAPT proposes the granularity which does not perform best. However, a loss of 1.6% is tolerable.

STAMP Benchmarks

For evaluation, we selected two benchmarks from the Stanford Transactional Applications for Multi-Processing (STAMP) suite [24]. In particular we choose

- `bayes`, an algorithm from the machine learning domain which learns the structure of a Bayesian network, and
- `kmeans` a data mining algorithm implementing the K-means clustering.

The chosen applications have different transactional characteristics: `bayes` exhibits long transactions, large read/write sets and high contention whereas `kmeans` has short transactions, small read/write sets and low contention [24]. The results of our static analysis are shown in Table 7.5. Performance changes are reported as relative improvements. The run time of the selected variant (named *rt_sel*) is compared with the variant that was not selected (named *rt_non*). The relative improvement is calculated according to $\frac{(rt\_non - rt\_sel)}{rt\_non} * 100$ because a lower execution time is better and, thus, an improvement. In both cases the algorithm proposes the cache line-based conflict detection granularity. This is correct despite the fact that the main data structures in memory were not analyzable statically. Due to the different input sets for the benchmarks, the programs read data at run time from files, hereby withdrawing from static analysis. However, there are relative performance improvements of 14.7% for `kmeans` and 16.9% for `bayes`.

7.2.3 Conclusion and Outlook for MAPT

This chapter presents a novel approach for detection and analysis of memory access patterns in transactions. The MAPT approach enhances the LLVM compiler framework and proposes an STM conflict detection granularity to the programmer. These mechanisms are evaluate with test cases and benchmarks from the STAMP benchmark suite. The results are promising and validate that an improved throughput can be achieved for the test cases as well as a reduced execution time is possible for the two benchmarks.

Further, the analysis pass shall be complemented with the results of a pointer analysis to address the limitations in the presence of function and data pointers.

8. First Experience with BG/Q Performance

The first experience with the performance of TM on the new BG/Q architecture is presented in this chapter and is similar to [171]. An introduction in Section 8.1 motivates the use of TM in high performance computing (HPC). Section 8.2 compares our approach with closely related work. Section 8.3 describes our experimental setup, the TM architecture of the BG/Q system, and our benchmark used to determine overheads. Section 8.4 presents low-level measurements, followed by the lessons learned in Section 8.5. Section 8.6 shows how we can use our lessons learned to add transactions to a Monte Carlo code and to a Smoothed Particle Hydrodynamics method. Section 8.7 summarizes the findings of the first experience with the BG/Q architecture.

8.1 Demands on Transactional Memory in HPC

Achieving efficient and correct synchronization of multiple threads is a difficult and error-prone task. The correct use of lock-based schemes requires a strict coding discipline to place matching lock and unlock operations into the code in a way that avoids race conditions and/or deadlocks. Additionally, lock-based synchronization often leads to high-overheads, either due to lock contention, when using coarse-grained locks, or unnecessary lock overhead, when using fine-grained locks. This not only slows down the process using the locks, but also has a global effect in large scale programming since it creates skew between processes as well as load imbalance, both major factors limiting the scalability of applications.

Transactional Memory (TM) has been proposed almost two decades ago to tackle these issues in shared memory systems [91]. TM simplifies synchronization by providing a single simple construct: the programmer wraps the critical instructions in a transaction (also called atomic block). These transactions are then executed optimistically in parallel and conflicting accesses are resolved by a TM run time system. As a consequence only the effects of entire and completed transactions are visible to concurrent threads, avoiding the visibility of intermediate memory states.

Except for a few, prototype implementations in research processors, TM has mainly been confined to software solutions and therefore has been burdened with significant

runtime overheads. These overheads severely restrict its applicability with the consequence that non-performance critical areas, for which the increase in programmability and ease of verification justify the additional cost, are the primary target. In high performance computing, however, the applicability of these approaches has been limited.

The recently introduced Blue Gene/Q (BG/Q) system by IBM for the first time provides Hardware Transactional Memory (HTM) in a commercially available platform [81]. BG/Q is designed as a large scale platform for scientific computing workloads. The first machine is installed at Lawrence Livermore National Laboratory and provides more than 1.5 million compute cores (with a total of 6 million hardware threads), making scalability one of the premier challenges on this machine.

This chapter presents the first comprehensive performance evaluation of BGQ's HTM capabilities from the application's perspective, although other recent papers have also measured certain aspects of performance [15] and [201]. Not every lock-based application will be suitable for HTM and it is important to understand what code properties lead to efficient executions and, hence, which codes can benefit from using HTM. In order to help code developers with this task, we provide a precise evaluation of the strengths and weaknesses of the architecture as well as what is required to map applications to the architecture in an efficient way. In particular, we focus on the synchronization primitives for parallel programming in shared memory architectures with OpenMP and provide detailed benchmark results. Our experiments take into account the application's characteristic (high or low contention), the influence of environment variables, the effects of enlarging transaction sizes, and hybrid parallelization with MPI. We apply our results to the optimization of a Monte Carlo Benchmark (MCB), which functions as a proxy application for several large scale Monte Carlo simulations, and a Smoothed Particle Hydrodynamics method from the PARSEC benchmark [35].

Specifically, we make the following contributions:

1. We introduce a new benchmark, CLOMP-TM, that is aimed at evaluating TM systems for scientific workloads.
2. We characterize the performance of HTM combined with OpenMP on BG/Q using CLOMP-TM.
3. We study the influence of thread count, environment variables and memory layout on TM performance.
4. We determine a fitting task to thread ratio for hybrid MPI/OpenMP codes with different synchronization primitives.
5. We identify code properties that are likely to yield performance gains with TM.
6. We condense the findings into best practices and apply them to a realistic Monte Carlo Benchmark code and a Smoothed Particle Hydrodynamics method.

For both case studies, an optimized TM version, executed with 64 threads on one node, significantly outperforms a simple or naive TM version validating the best practices derived from our observations with CLOMP-TM.

8.2 Comparison with Related Work

The only paper describing an early experience with a commercial hardware transactional memory implementation published in a major conference, to our knowledge, is by Dice et

al. [53]. The paper describes and evaluates the hardware transactional memory feature of SUN's Rock processor [33], which is no longer available, and focuses on the evaluation of concurrent data structures such as Red Black trees and `Hashtable`, and the construction of a minimum spanning forest [109]. The parallelization of these codes uses threads only. Thus, no experiments are made that estimate the performance of a hybrid parallelization with MPI. Further, there is an important difference between the HTM implementations of Rock and BG/Q. Rock is a checkpoint-based architecture which is exploited in the context of TM to save and restore the architectural state of the registers in hardware. In BG/Q the TM runtime performs this task in software, while only conflict detection is done in hardware. This important difference will affect the performance of both architectures and makes transferring results of previous studies from the Rock to the BG/Q architecture extremely difficult.

A description of a second commercial HTM implementation can be found in a paper by Click [42]. The goal is to accelerate the `synchronized` keyword in `Java`. Thus, no extensions to the language are made and explicit programming with transactions is not possible. Instead a heuristic decides whether to run a critical section as a transaction or not.

Most STM papers use STAMP, a benchmark suite for transactional memory research [24]. The codes comprise: Bayesian network learning, gene sequencing, network intrusion detection, K-means clustering, maze routing, graph kernels, a client/server travel reservation system, and Delaunay mesh refinement. This covers many application areas in which STMs have been used, but do not represent codes from the area of high performance computing, for which HTM is a promising approach to overcome synchronization overheads and to improve scaling of hybrid thread/MPI codes. In this work, we therefore focus on a new benchmark explicitly designed to cover this area and present results that demonstrate how HTM can be deployed in HPC.

8.3 Experimental Setup with BG/Q

For all following experiments we use an early prototype of BG/Q installed at IBM. TM is available through IBM's XL C/C++/Fortran Compiler suite for BG/Q, which provides new language primitives that allow users to specify transactions.

8.3.1 Overview of BG/Q's TM Hardware

The BG/Q prototype we had access to had 32 nodes with 16 cores each. Each core can execute up to four hardware threads. Transactional memory is implemented within the L2 cache, which consists of 16 banks of 2 MBytes each located across a full crossbar from the 16 compute cores. The L2 cache has a cache line size of 128 Bytes. Memory accesses that can lead to conflicts between transactions, are tracked by the L2 cache, which is the point of coherency. Conflict detection between different transactions is completed in hardware, while conflict resolution is coordinated through the TM software stack. Note that, in addition to TM, the L2 cache also implements an improved set of atomic operations that targets faster thread synchronization. Comparisons in the remainder of the chapter between TM and atomic operations therefore provide results between two novel and highly optimized schemes. More information on BG/Q's hardware in general can be found in a recent presentation by Haring at Hot-Chips [81].

8.3.2 Application Perspective in BG/Q's TM Software Stack

By default, the TM runtime uses a *lazy* (or optimistic) conflict detection scheme at commit time, as the runtime suppresses the hardware from sending interrupts to the conflicted threads as conflicts are detected. However, applications/users can enable a *pessimistic detection* scheme by setting the `TM_ENABLE_INTERRUPT_ON_CONFLICT` environment variable. That is, conflict arbitration happens immediately at the time of conflicts. Either scheme needs to be carefully chosen as an already doomed thread, if allowed to finish, may cause further spurious conflicts.

The TM runtime also relies on a lazy versioning (i.e., write-back) scheme as all speculative writes are buffered in the multi-versioned cache until commit time. Strong atomicity (i.e., opacity) is guaranteed unless a thread is running in irrevocable mode. In such a case, the thread runs non-speculatively and all writes take effect immediately. The `TM_MAX_NUM_ROLLBACK` environment variable controls when a thread should enter into irrevocable mode. The irrevocable mode is a mechanism that guarantees that a thread makes progress. The contention manager favors an older thread to commit based upon the timebase register value of the thread at the time when speculation starts. Aborting a transaction does not back-off for a pre-determined time, rather, a thread retries immediately. The runtime also implements flat nesting whereby commits and rollbacks are to the outermost TM region. As an additional feature, the runtime monitors the TM behavior of the application and provides the resulting TM statistics to the user. All TM statistics presented in this work, are retrieved by this method.

8.3.3 The CLOMP-TM Benchmark

Since current TM benchmark suites do not provide the necessary coverage for scientific applications, we focus on the development of a new benchmark, specifically designed to expose the range of properties needed to characterize scientific workloads: CLOMP-TM[1]. It is designed to compare multiple synchronization constructs. In fact, it was created to help us mimic the application characteristics of several large scale, multi-physics applications used in production at DOE laboratories. It achieves this flexibility and wide coverage through a series of parameters that allow us to explore varying transaction granularities and conflict rates, coupled with typical computational kernels found in scientific codes.

CLOMP-TM originates from the publicly available CLOMP benchmark [21] used for evaluating OpenMP implementations. CLOMP resembles an unstructured mesh with a set of partitions. Each partition holds a linked list of zones. To vary the pressure on the memory system, the size of these zones can be configured. Computation, that is performed when updating a zone, only uses the first 32 Bytes of each zone. The computation per zone update can be scaled by a factor. While the original CLOMP aims to quantify overheads due to threading and the specific OpenMP implementation, the CLOMP-TM aims at quantifying and comparing synchronization overheads of multiple synchronization constructs. CLOMP-TM can be configured to resemble the synchronization characteristics of typical scientific applications used in HPC. Thus, performance results of CLOMP-TM not only enable us to provide detailed characteristics of the low-level properties of the BG/Q, but also, and more importantly, to project the performance impact of TM on large scale parallel applications.

[1]For the experiments CLOMP-TM version 1.59 is used and made available at `https://asc.llnl.gov/sequoia/benchmarks/#clomptm`.

Name	Description	Contention
None	Threads do not conflict.	No contention.
Adjacent	Updates of adjacent memory addresses.	No to small contention.
Random	Repeatable but randomly seeming updates.	High contention.
FirstParts	Only the first parts are updated.	Highest contention.

Table 8.1: Different contention levels in the CLOMP-TM benchmark (cf. to [171]).

Name	Implementation	Description
Bestcase	—	Bestcase without synchronization.
Serial Ref	—	Serial reference implementation.
Small TM	`#pragma tm_atomic`	Synchronizing each update with a transaction.
Small Atomic	`#pragma omp atomic`	Synchronizing each update with an atomic operation.
Small Critical	`#pragma omp critical`	Synchronizing each update with OpenMP's critical section.
Large TM	`#pragma tm_atomic`	All scatter zone updates in one transaction.
Large Critical	`#pragma omp critical`	All scatter zone updates in one critical section.
Huge TM X	`#pragma tm_atomic`	X times *Large TM* in one transaction.

Table 8.2: Description of synchronization constructs used in CLOMP-TM [171].

Major changes over CLOMP

In order to study the impact of TM in the presence of loop dependencies and the resulting conflicts, CLOMP-TM adds explicit and controllable dependencies to the loop structures of CLOMP. Besides the implementation with TM, CLOMP-TM also tests optimistic execution not secured by any synchronization[2] or using other constructs such as atomics or OpenMP-based constructs with the same level of abstraction in terms of programming.

For a meaningful comparison of optimistic and pessimistic synchronization constructs, multiple memory access patterns have to be considered. These memory access patterns determine the likelihood of a conflict between concurrent accesses of two threads. A single parameter defines the zones that are updated by a thread. The contention arises when multiple threads update the same zones. These different contention scenarios are shown in Table 8.1.

In comparison to the CLOMP benchmark, the updates of a zone are enlarged. This new construct is called *scatter zone* and enables larger critical sections, which resembles the update of multiple variables (e.g., coordinates with multiple dimensions x, y, and z) in one critical section. For the *large* versions of the synchronization constructs, the parameter *scatterCount* defines the number of updated zones in a single synchronized block.

[2]In this configuration CLOMP-TM does not return correct results, but the timings can be used to study conflict free cases.

Each iteration executes the selected computation pattern. Available patterns with increasing complexity are: *none*, *divide*, *manydivide*, and *complex*. *None* performs no computation, *divide* performs a single, *manydivide* multiple, floating point divide operations and *complex* exercises math functions e.g., `log` and `sqrt`. CLOMP-TM is carefully designed to eliminate as much noise as possible: I/O is performed only outside of timing loops and all the loops are through with several iterations just before the timing loops to eliminate start up costs and cold cache effects. Table 8.2 holds the synchronization constructs to be compared.

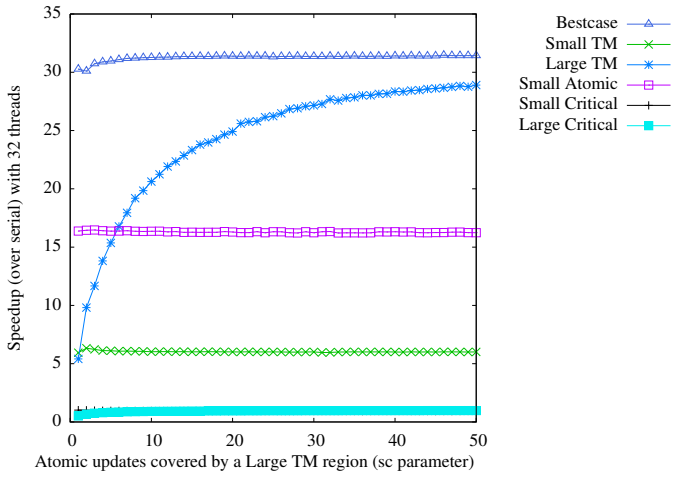
Comparison of CLOMP-TM with TM Benchmarks

Apart from the heavily cited STAMP benchmark suite [24] that does not represent the scientific application behavior we are interested in, a growing number of parameterized workloads gains popularity. An important example is the WormBench workload [218]. WormBench is derived from the popular snake game and has been designed to evaluate and verify the efficiency of a TM system. WormBench is written in *C#* and enables performance comparisons between TM and global locks. Parameters are size of the world (matrix), number of worms (threads), body and head size of a worm and operations to be performed while moving. In contrast, CLOMP-TM enables a comparison with a single instruction atomic update, an unsynchronized version and two sizes of transactions and critical sections, respectively. For our workloads *C#* does not play an important role and we believe that a more versatile benchmark to model scientific workloads is needed, which lead us to develop CLOMP-TM. A more suited candidate for the modeling of arbitrary TM workloads is Eigenbench [94]. Eigenbench uses orthogonal metrics to model a specific workload. For our use case, not knowing *a priori* how TM will perform, we would have to rewrite the application with transactions, measure the metrics, derive a configuration for Eigenbench and use this to model our workload. With CLOMP-TM, the user only needs to know the number of shared memory accesses and the number of floating point operations per loop iteration to set the right parameters that will resemble the application behavior.

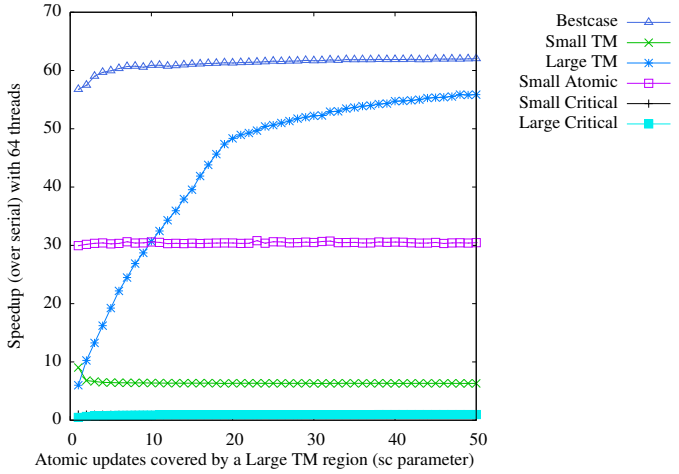
8.4 Characterizing TM Performance using CLOMP-TM

For the CLOMP-TM cases presented in this work, synchronization overheads can dramatically affect speedup. We vary the parameters of CLOMP-TM to learn how the parameters affect the speedup and to find out what code properties qualify for TM. These results will help application developers to tune their codes (e.g., through picking a better suited synchronization primitive), but the achievable speedup is determined by the properties of the application (e.g., ratio of computation and synchronization, contention for memory locations).

For our initial experiments targeted at understanding the potential for TM, we chose the parameters for CLOMP-TM in a way that TM outperforms a highly efficient implementation of `omp atomic`. In this configuration, CLOMP-TM performs 8 divide operations per zone update with a stride of 4. Threads do not contend for memory locations. We increase the size of the scatter zone so that an increasing amount of updates are carried out in *Large TM*. Figure 8.1 a) illustrates that in the case of 32 threads performing 5 zone updates is the cross-over point for *Large TM* over *Small Atomic*. For 64 threads the number of zones is twice as high (see Figure 8.1 b)). Please note that the large amount of computation per zone update masks the overheads of synchronization.



(a) 32 threads.

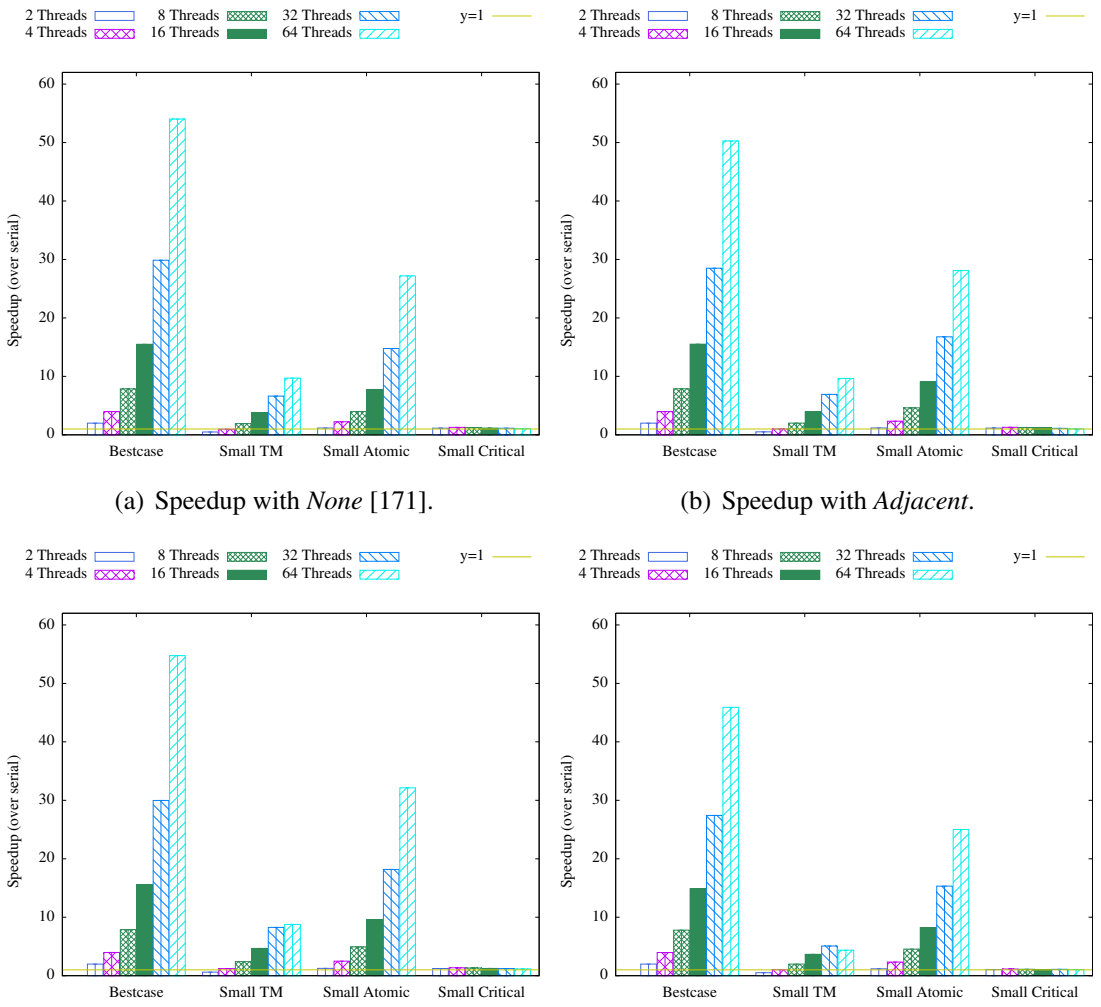


(b) 64 threads.

Figure 8.1: CLOMP-TM performing 8 divide operations with a stride of 4 per zone update with excellent speedups of *Large TM* over *Small Atomic*. Run with `clomp-tm-bgq-divide4 -1 1 256 128 256 stride1,1,stride1%/2 sc 1 0 6 100`. First published in [171].

numParts	64
zoneSize	128
zone alignment	128
scatter	3
flopScale	1
timeScale	100
Zones per Part	100
Total Zones	6400
Zone Calc Stride	1
Extra Zone Calcs	8
Zone Calc Flag	-DDIVIDE_CALC
Zone Calc Formula	((1.0/(x+2.0))-0.5)

Table 8.3: Parameters for CLOMP-TM; similar to [171].



(a) Speedup with *None* [171].

(b) Speedup with *Adjacent*.

(c) Speedup with *Random* [171].

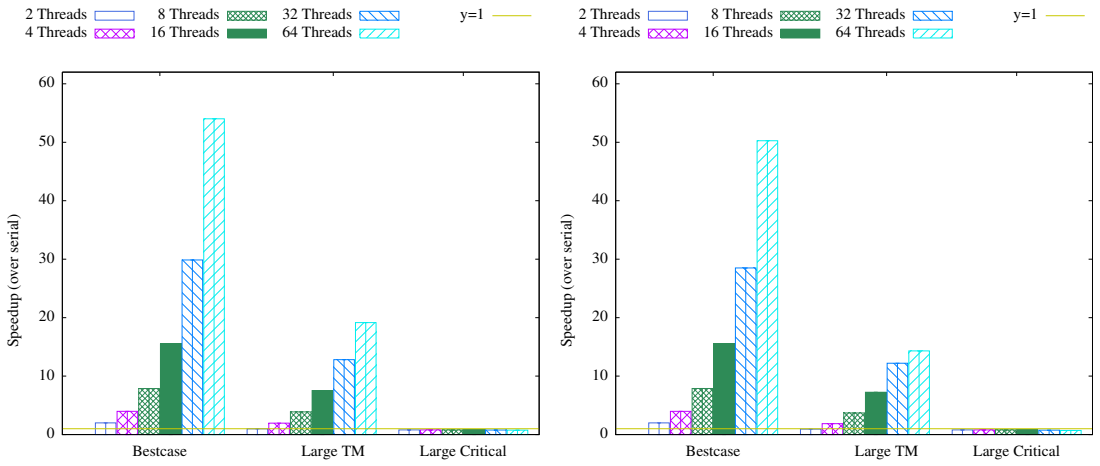
(d) Speedup with *firstParts*.

Figure 8.2: CLOMP-TM performing 8 divide operations per zone update with small critical sections [171].

8.4.1 Synchronization Overhead

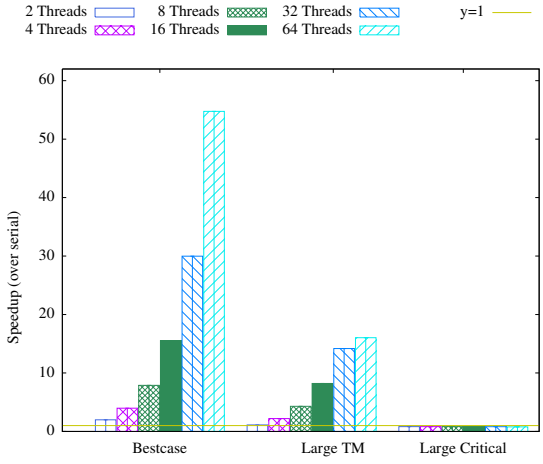
To understand the tradeoffs in using TM, we first study the synchronization overhead associated with different approaches and contrast them to the TM results. We obtain the results in this section by using the parameters shown in Table 8.3. Memory is allocated by the main thread. This is sufficient because memory accesses are uniform in BG/Q. The setting of *zonesPerPart* equal to 100 stems from the original CLOMP and mimics the loop sizes of many multiphysics applications [21]. The chosen computation pattern is *divide*. For each zone update 8 extra *divide* calculations are executed. The environment variable `OMP_WAIT_POLICY` has been set to `ACTIVE` for all runs.

Figure 8.2 shows the speedup of the different synchronization mechanisms for updating one memory location at a time. *Small Atomic* achieves the highest speedup for a small critical section compared to *Small TM* and *Small Critical*. This relation holds for the access patterns *None* that generates no conflicts and *Adjacent* that only generates conflicts for more than 16 threads. For smaller thread counts, the zones, that each thread updates, do not overlap between threads. This *Adjacent* pattern is an interesting case that illustrates that the programmer must be very careful when designing the memory layout of the

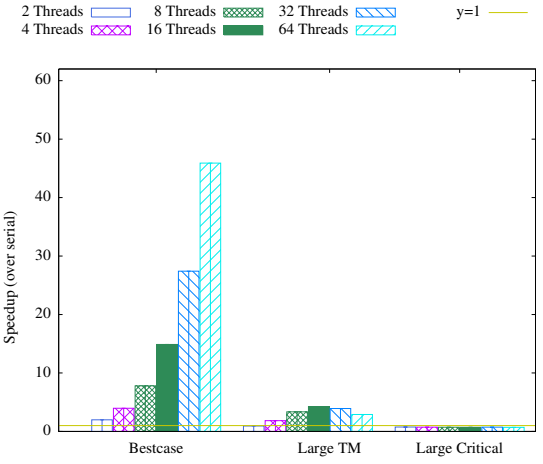


(a) Speedup with *None* [171].

(b) Speedup with *Adjacent*.



(c) Speedup with *Random* [171].



(d) Speedup with *firstParts*.

Figure 8.3: CLOMP-TM performing 8 divide operations per zone update with large critical sections (cf. to [171]).

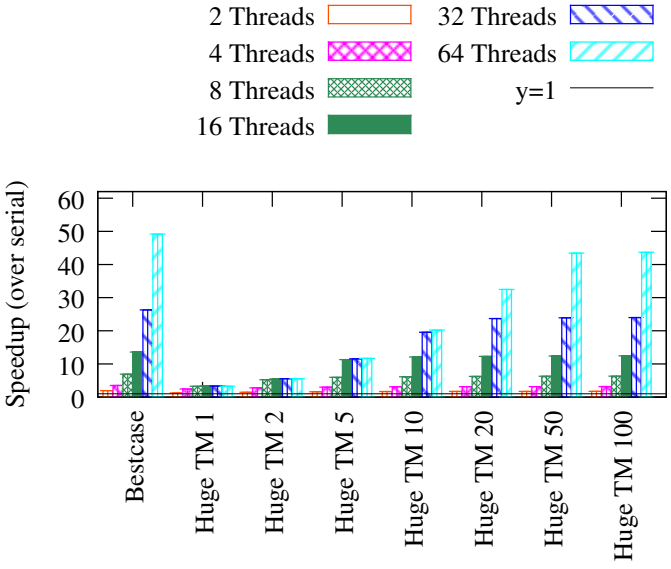


Figure 8.4: CLOMP-TM (v. 1.36) performing 8 divide operations per zone update with huge critical sections. Speedup shown with *None* [171].

application with respect to the sharing between threads. Also, with contention, generated by the *Random* and *firstParts* memory access patterns, *Small Atomic* provides the fastest synchronization.

Large TM outperforms *Large Critical* as can be seen for no and high contention cases of Figure 8.3. Thus, for critical sections with more than one memory update, TM is the preferred method. The *Huge TM* with 100 times the size of *Large TM* performs extremely well in case of no contention (see Figure 8.4). Further experiments with higher contention cases reveal that this speedup is very fragile. These experiments demonstrate (and the TM statistics confirm) that longer transactions are more susceptible to contention and should be deployed with great care.

8.4.2 Conflict Probability

The performance of any TM application depends on the number of conflicts it has to encounter that lead to potentially costly rollbacks. We study this issue by extending CLOMP-TM with a special mode that enables transition between scatter modes. Thus, a parameter has been added that defines the number of *intended conflicts* for a run. For our experiments we compute this parameter from a *conflict probability* (cp) as follow: $total\ zones * scatter * cp$. Updates are counted as *intended conflicts* and performed inside a large transaction. Note, however, that not all *intended conflicts* lead to an actual conflict and some conflicts can cause multiple rollbacks.

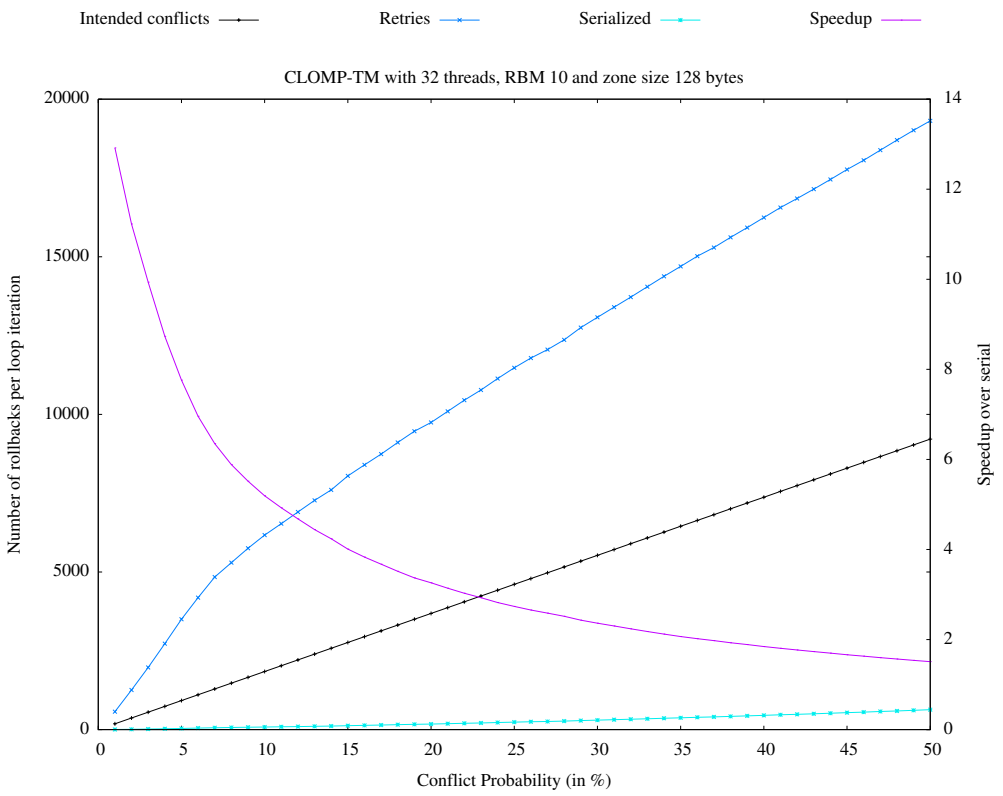
Figure 8.5(a) illustrates the impact of the number of retries on the achievable speedup. We can clearly see that a linear increase of the conflict probability (shown as intended conflicts) leads to an exponential decrease of the speedup. In this experiment the zone size is set to 128 Bytes. In case it is smaller (e.g., 64 or 32 Bytes) conflicts may be falsely detected because two zones are mapped to the same cache line. These *False Positives* are eliminated when the zone size equals the size of the L2 cache line.

8.4.3 Tuning the BG/Q TM Runtime Environment

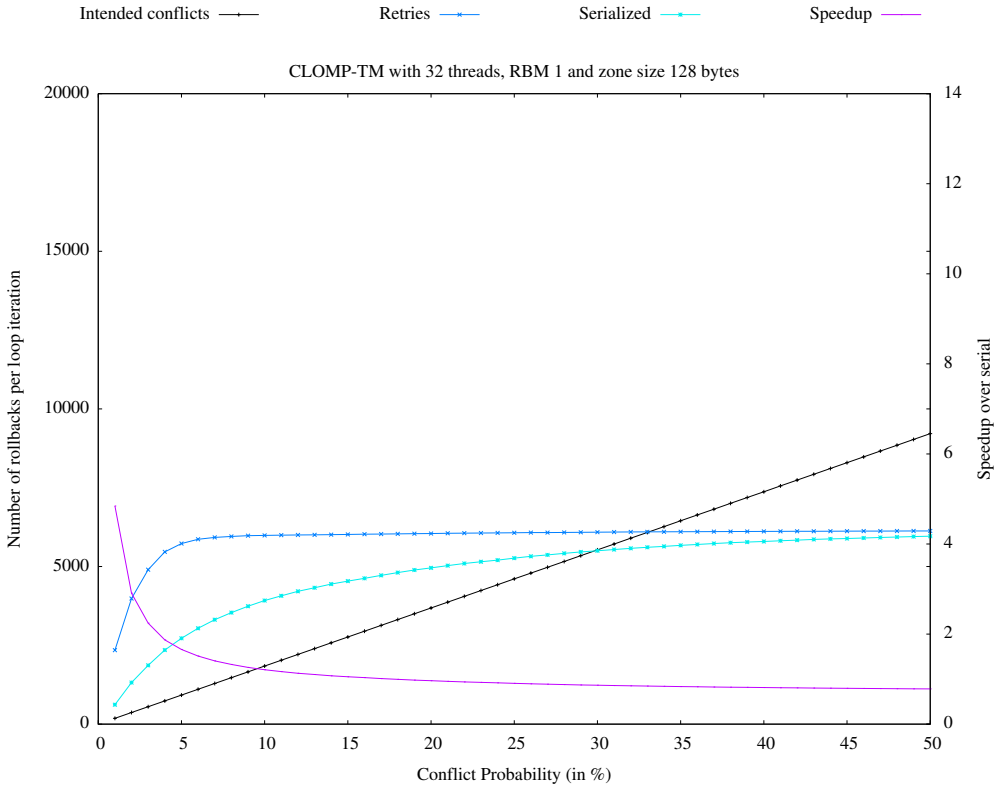
The TM runtime system in BG/Q provides a series of parameters that can be used to fine tune the performance of HTM applications. These parameters are available to the user through environment variables that can be set before the code's execution. The most significant one is `TM_MAX_NUM_ROLLBACK` (RBM), which controls the number of times a transaction can be aborted and rolled back before the runtime gives up on it and executes it in *irrevocable mode*, i.e., the transaction is executed non-transactionally under a global lock so that other transactions can not interfere. The TM runtime will further mark this transaction and execute it in irrevocable mode right away on a subsequent execution.

Figure 8.5 shows results with 32 threads and RBM set to 1 and 10. In Figure 8.5(a) RBM is set to 10 and shows a significant higher number of retries than Figure 8.5(b) (RBM 1). The relative number of serialized transactions is higher for RBM 1. Both observations are due to the RBM setting, since a smaller RBM value serializes after fewer retries. In terms of speedup, RBM 10 outperforms RBM 1 due to less frequent serialization. The effects of the adaptive TM runtime on the performance need closer investigation.

Figure 8.6 demonstrates the influence of RBM 1, RBM 5 and RBM 10 on the achievable speedup with TM. For all contention levels and *Small TM* the differences between RBM 5 and RBM 10 are insignificant. For higher contention and *Large TM*, RBM 10 has slight



(a) 32 threads with RBM 10.



(b) 32 threads with RBM 1.

Figure 8.5: Influence of `TM_MAX_NUM_ROLLBACK` (RBM) on retries/speedup. Run with `clomp-tm-bgq-divide1 -1 1 x1 d6144 128 firstZone,`*cp*`,randFirstZone 3 1 0 6 100`. First published in [171].

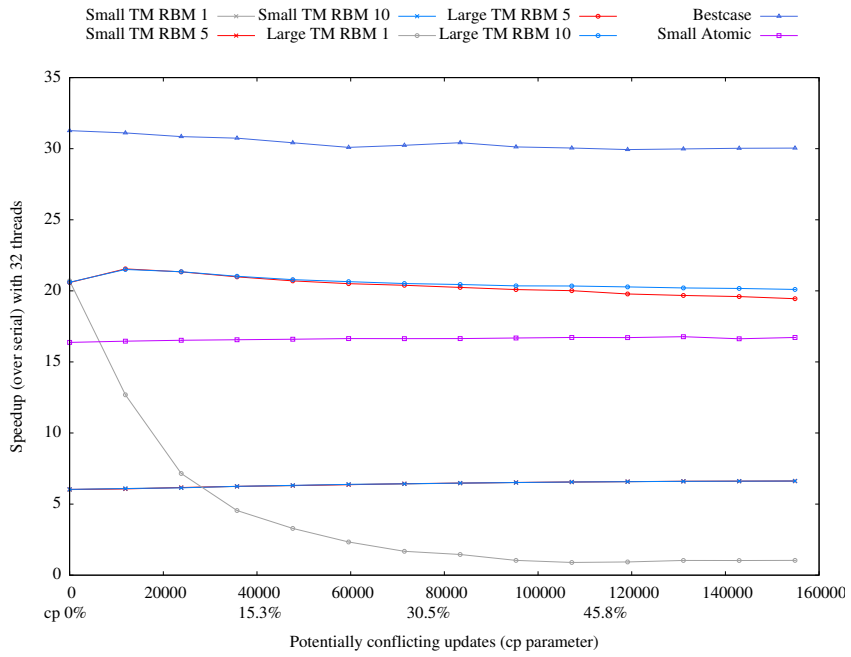


Figure 8.6: Studying the influence of setting `TM_MAX_NUM_ROLLBACK` with CLOMP-TM and changing the level of contention [171]. Run with `clomp-tm-bgq-divide4 32 1 256 128 256 stride1,`*cp*`,stride1%/2 10 1 0 6 100`.

advantages over RBM 5. RBM 1 shows the worst performance for the presented level of contention.

A second important parameter for TM is the scrub rate. It triggers a garbage collection for TM SpecIds, which mark entries in the cache as belonging to the same or different transactions. Figure 8.7 shows that varying the scrub rate has a large impact. For our benchmark with a lot of transactions and short intervals between these, a scrub rate of 6 is best.

In the next paragraph, we will research another important aspect: the influence of the scrub rate on the number of rollbacks. Figure 8.8 shows the number of rollbacks of *Small TM* and *Large TM* on the y-axis and the scrub rate on the x-axis. All four plots show that the variation in rollbacks depends on the size of the transaction for this parameter setting. For *Large TM* the influence is higher (but for *Large TM* the total amount of rollbacks is higher). For *Small TM* the rollbacks are almost always zero except for occasional peaks for one setting of the scrub rate. This is an interesting phenomenon because it shows the random influence that changing the scrub rate can have through randomly delaying threads (cf. to Figure 8.8(b)). For *Large TM* and 4 threads the highest conflicts occur with the highest scrub rate but the relative differences are not high. Figure 8.8(b) and Figure 8.8(c) show for *Large TM* with 8 and 16 threads respectively that small values for the scrub rate can yield less conflicts than higher values. An explanation is that scrubbing may even prevent some conflicts because it blocks access to the L2 cache and thus other threads from proceeding. For *Large TM* and high thread counts (32 and above) the trend is that a higher scrub rate yields more conflicts (see Figure 8.8(d)). These findings help to understand the various ways in which the setting of the scrub rate combined with the number of threads influences the TM behavior of the application.

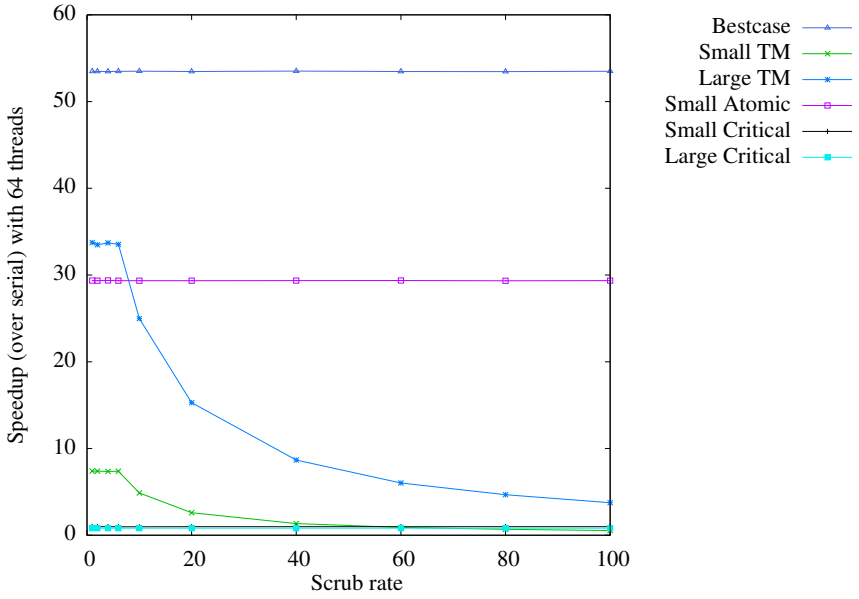
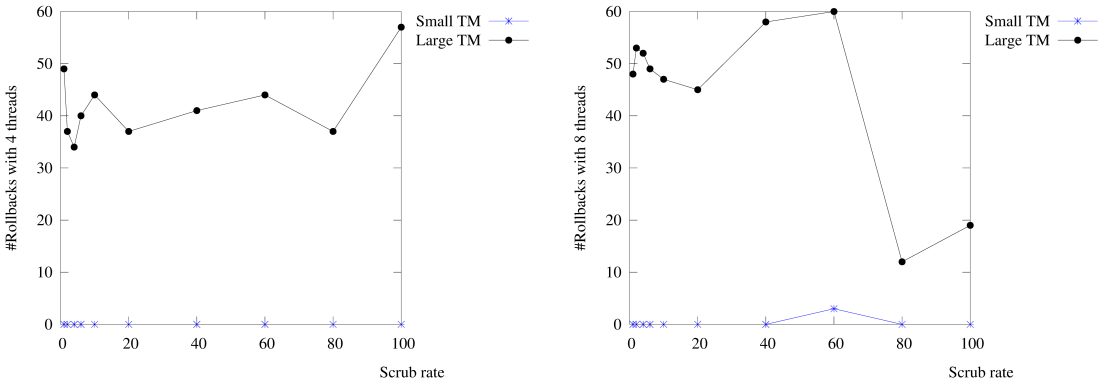
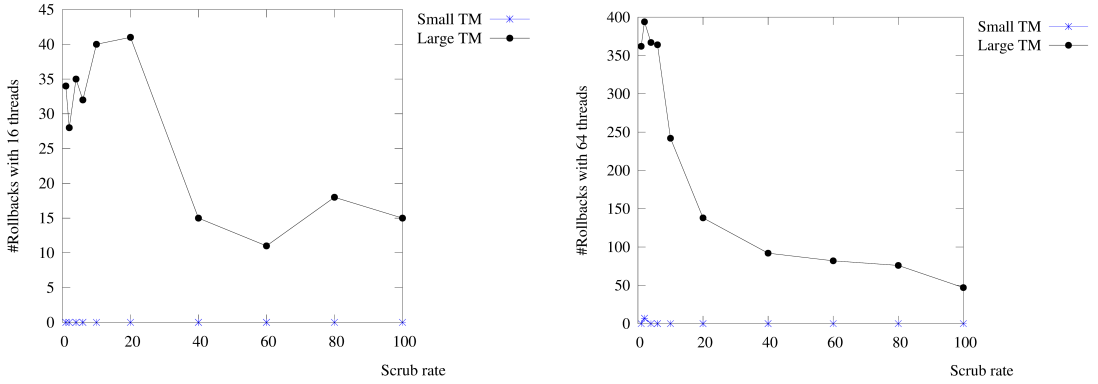


Figure 8.7: Influence of the scrub rate for SpecIds on performance with 64 threads [171]. Run with `clomp-tm-bgq-divide1 -1 1 64 100 128 InPart,10,firstParts 10 1 0` *sr* `100`



(a) 4 threads. (b) 8 threads.



(c) 16 threads. (d) 64 threads.

Figure 8.8: Influence of the scrub rate for SpecIds on the amount of rollbacks. Run with `clomp-tm-bgq-divide1 -1 1 64 100 128 InPart,10,firstParts 10 1 0` *sr* `100`

```
1 MPI_Barrier(MPI_COMM_WORLD);
2 get_timestamp (&bestcase_start_ts);
3 do_bestcase_version();
4 get_timestamp (&bestcase_end_ts);
5 MPI_Barrier(MPI_COMM_WORLD);
```

Listing 8.1: Use of MPI barriers for CLOMP-TM with MPI.

8.4.4 CLOMP-TM with Mixed Scatter Modes

So far, we have only discussed settings with a single scatter mode at a time (see Table 8.1). This leads to a fixed TM application behavior that defines the contention between threads for the whole program run. As a result, TM either performs excellent because of the lack of conflicts (e.g., scatter mode None) or suffers from the frequent retries (e.g., firstParts). This model, while useful to get point results, is too restricted to model all scientific workloads and expose the potential of TM. CLOMP-TM therefore additionally supports two different scatter modes that execute alternately. It uses a parameter to define how often the second scatter mode will be used, i.e., increasing this parameter leads to more updates with the second scatter mode.

8.4.5 Using TM in the Context of MPI Applications

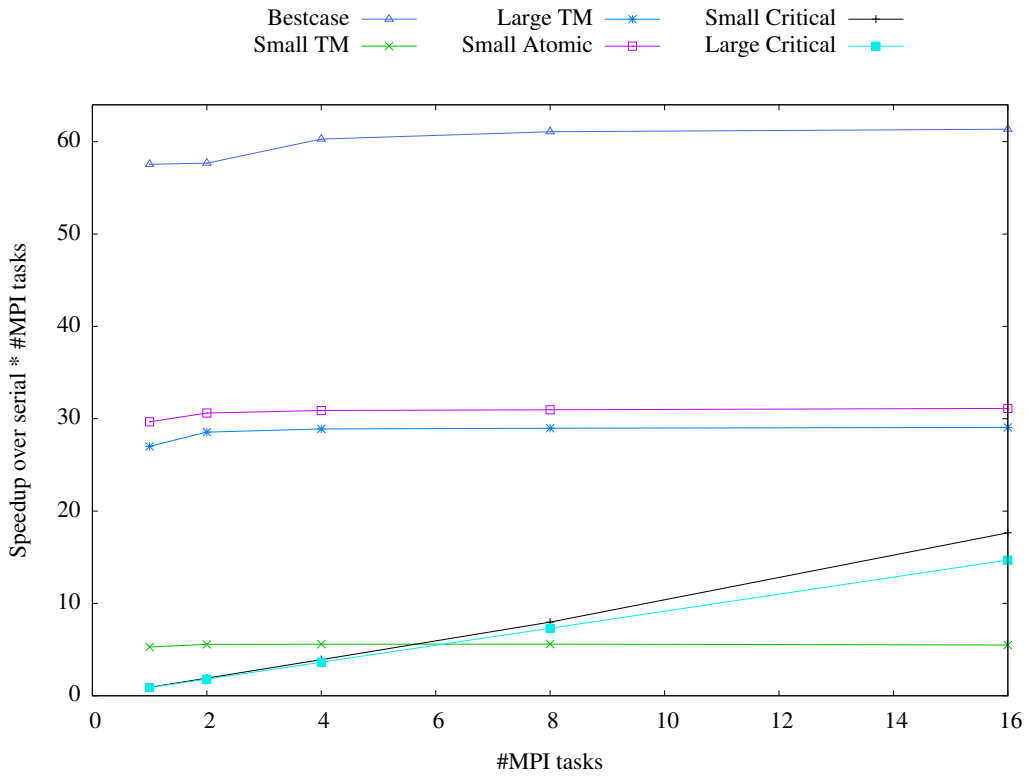
Up to this point, we focused on single node experiments using OpenMP as the method for threading. To work across nodes and hence to exploit the vast parallelism available in BG/Q systems, scientific applications will require additional parallelization with MPI. Consequently, it is important to understand the interplay between OpenMP threading with TM support and having multiple MPI tasks on the node.

In the following we study the side effects of running multiple MPI tasks, each executing CLOMP-TM, on one node. Our goals are:

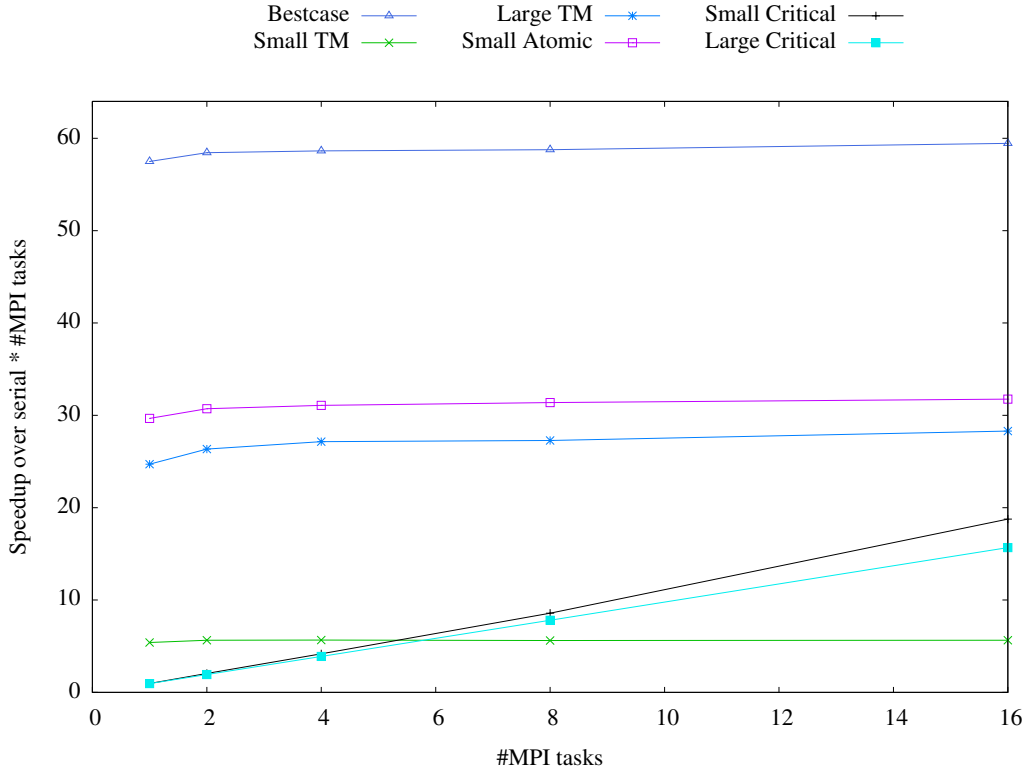
1. to verify the robustness of the results above,
2. identify bottlenecks due to the sharing of architectural resources,
3. and determine a fitting MPI task to OpenMP thread ratio.

We extended CLOMP-TM to execute multiple instances of its core functionality, synchronized by MPI operations. Besides calls to init and finalize MPI, we inserted *MPI_Barrier*s. These barriers are placed such that all MPI tasks execute the code for the same synchronization primitive. An example for the placement of the *MPI_Barrier* calls is shown in Listing 8.1. To execute in this lock step fashion guarantees that all MPI tasks execute the code for the same synchronization primitive. As a consequence, we can directly control the contention on the architectural resources that are necessary for synchronization (such as the L2 cache). Thus, the methodology provides a clear and controllable mechanism to study the architectural resources needed by individual synchronization primitives.

Figure 8.9 illustrates the performance of CLOMP-TM with MPI for small and large critical sections using *strong scaling*, i.e., the total amount of work is constant for all task counts. We achieve this by dividing the number of parts (initially 1024) as well as the number of updates in the second scatter mode by the number of MPI tasks. All MPI tasks execute as many threads as possible without oversubscribing the node (e.g., 1 MPI task executes 64 threads).



(a) No to low contention (rollback per transaction ≈ 0).



(b) High contention (rollback per transaction ≈ 1).

Figure 8.9: CLOMP-TM with MPI performing 8 divide operations with a stride of 4 per zone update with no and high contention [171]. Run with `clomp-tm-mpi-bgq-divide4 -1 1 (1024/taskno) 128 256 stride1,`*cp*`,stride1%/2 10 1 0 6 100`. *cp* is set to $\frac{16}{taskno}$ for the left and $\frac{1.12*10^6}{taskno}$ for the figure on the right.

Figure 8.9(a) shows the average speedup of the threads in each MPI task multiplied by the number of MPI tasks on the y-axis. The number of MPI tasks is plotted on the x-axis. In this case with extremely low contention, *Large TM* performs almost as well as *Small Atomic*. The surprise is that for large task counts, *Small* and *Large Critical* perform better than *Small TM*. Especially *Small Critical*, which is protecting one memory location, has been optimized heavily in the new BG/Q L2 cache and is now a strong alternative for TM if the granularity in the codes allows for this. With increased contention (Figure 8.9(b)) *Large TM* and *Small TM* perform slightly worse but the overall trend with respect to the critical section remains. *Large Critical* benefits from the smaller thread numbers at higher task counts because the cost for serialization is reduced.

8.4.6 Finding a Competitive Task to Thread Ratio

The architecture of one BG/Q node features 16 compute cores each equipped with 4-way hyper-threading. As demonstrated in earlier papers [21], an OpenMP barrier has a higher overhead for higher thread counts. Thus, a hybrid parallelization with MPI and OpenMP may achieve higher performance than an OpenMP only implementation. In order to be able to compare results of OpenMP and hybrid parallelization, we use a simple metric. For the hybrid case, we multiply the reported OpenMP speedup with the number of MPI tasks. Figure 8.9 shows that the *Bestcase* across MPI tasks is stable. Across all tested memory access patterns and MPI tasks configurations, the OpenMP version with the highest possible thread count performs best. While this is not surprising since the BG/Q architecture requires at least two threads per core to be able to reach full instruction issue bandwidth, it is nevertheless an important first insight that we gain from this experiment. For architectures with hyper-threading, the additional HW threads are often turned off because they lead to a slowdown. For the BG/Q architecture running the CLOMP-TM (with MPI) benchmark this is not the case. Every thread (even beyond the minimum of two needed for full issue bandwidth) contributes an important part of the reported performance. For the executed strong scaling experiments, however, the results of finding a preferable task to thread ratio are inconclusive. All tested ratio perform well and differences are extremely small.

In all high contention cases *Small Atomic* performs best. For cases with little to no contention *Large TM* may perform almost as well as *Small Atomic*. The large transactions benefit from the optimistic concurrency and the overhead for setting up the transaction is amortized due to the long transaction size. Unfortunately, this effect is limited to scenarios where expensive roll back operations are infrequent.

8.5 Lessons Learned

The experiments described above give us a clear characterization of HTM on BG/Q and provide the necessary information to understand which kind of applications can benefit from HTM. In the following we summarize these findings in a set of best practice guidelines that will help code developers on BG/Q decide if and how to best exploit HTM.

In particular, codes that exhibit the following properties are likely candidates for HTM:

- critical section should have low contention so that conflicts are unlikely,
- critical sections should access more than one memory location (preferably in the range of 10 to 20) so that `omp atomic` is not applicable and TM's property of providing atomicity for updates of multiple memory locations is valuable,

- high computation to synchronization ratio so that computation can mask the overheads of synchronization.

For synchronization with OpenMP, both the size of the code region that needs to be executed atomically and the potential conflict rate play an important role:

- For code regions that only require atomic updates using one instruction, *omp atomic* shows the best performance, since it can be mapped to the efficient atomic instructions implemented in the BG/Q L2 cache.
- For larger critical sections with low to moderate contention and conflict potential ($<<$ 1 rollback per transaction), TM using the *tm_atomic* primitive is beneficial, since the costs of conflicting transactions are amortized by avoiding serialization.
- In case of very high contention ($>$ 1 rollback per transaction) and small critical sections, *omp critical* also outperforms TM, since TM conflicts and rollbacks start dominating leading to higher overhead.
- For applications that are not utilizing the full memory bandwidth with a high transactional execution time and short times in between transactions, setting the scrub rate to 6 yields better performance.

These findings complement a previous study on using Software Transactional Memory for scientific codes using a different and more specific setup [14]. Additionally, researchers already identified codes that match the criteria from above and are expected to benefit from TM [208]. This work was limited to STM methods and has only recently been verified on a HTM system, publishing first performance results of the BG/Q architecture [15]. Our current recommendations verify the applicability of these previous preliminary studies to HTM, extend them by adding tradeoffs offered by the new adjustable performance parameters found in IBM's HTM solution, and generalize them to a more comprehensive guide for application developers.

8.6 Application Case Studies

8.6.1 MCB: A Proxy Application for Monte Carlo Simulations

In this section we apply the best practices from the previous section to a benchmark closely representing a real world application. The Monte Carlo Benchmark (MCB) models a Monte Carlo simulation, a popular technique for physics simulations. In contrast to classical simulation approaches, Monte Carlo simulations do not compute their result explicitly, but instead adaptively sample the simulation domain and execute individual simulations for each sample. This process is repeated until the probability of a result can be quantified.

The initial MCB code was already parallelized with MPI and OpenMP using *omp critical* and *omp atomic* to synchronize OpenMP threads (denoted as *Critical & Atomic* in the following). As a first, naive TM implementation, referred to as *TM naive*, we replace all critical sections with transactions and set the TM environment variables to their default for *TM simple* and *Critical & Atomic*.

Additionally, we create an optimized version, called *TM opt*, following the lessons in the previous section. In *TM opt* we use a hybrid strategy matching the characteristics for each synchronization construct: synchronizations that involve only one instruction use *omp atomic*, while all *omp critical* constructs are replaced with *tm_atomic*.

Code version	*Critical & Atomic*	*TM naive*	*TM opt*
Speedup	27.57	20.06	27.45

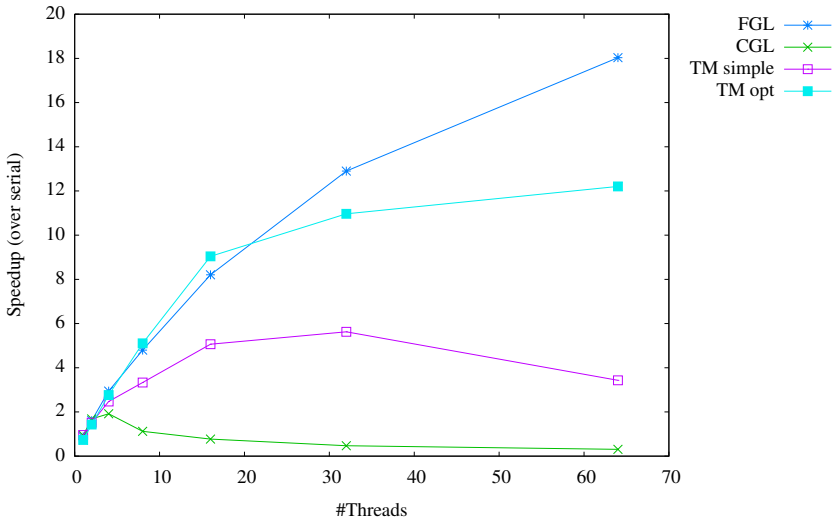
Table 8.4: MCB with one MPI task and 64 threads (strongScaling) – speedup over baseline.

Table 8.4 shows the results of the experiments with one MPI task and 64 threads in a strong scaling experiment with $5 * 10^6$ particles. Each value is an average of "samples per second" over 30 runs and normalized to baseline: "samples per second" with one MPI task and one thread. *TM opt* has a speedup over baseline of 27.45 and performs almost as good as the original version, but at reduced code complexity and programmer effort. The result of *TM naive* demonstrates that the lessons learned in this work are essential to getting good performance. Further experiments reveal a limited potential for optimizing the synchronization of threads in MCB. Commenting out all occurrences of `omp atomic` and `omp critical` (and ignoring the fact that this results in wrong answers for the simulation) yields $\approx 5\%$ performance improvement.

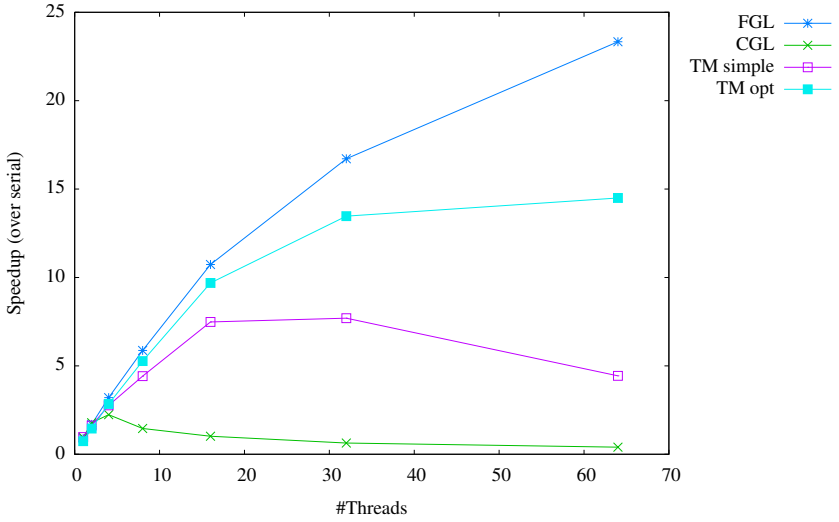
8.6.2 Fluidanimate from the PARSEC Suite

In addition to the Monte Carlo Benchmark, we use `fluidanimate` from the PARSEC benchmark suite [35]. `fluidanimate` implements a Smoothed Particle Hydrodynamics (SPH) method to animate fluid dynamics. To include it in the PARSEC suite, the application has been parallelized with Pthreads and fine-grained locking. Under the assumption that particles can not travel more than one cell in one time step, this parallelization uses an array of locks to protect the boundaries. An if-statement checks whether a lock needs to be taken. This synchronization pattern is rather sophisticated and by far exceeds the complexity of a single global lock. Because the programming complexity of TM can be compared with a single global lock, we added two versions: one with a coarse grain lock (cgl) and a simple TM version (TM simple) that replaced all lock acquire and lock release operations (that occur in three code segments) with a transaction.

Then, we apply the lessons learned from Section 8.5. First, we enlarge all three transactions by removing the if-statement and changing the two outer loops to be inner loops. More measurements reveal that for the third transaction having three nested inner loops is best. Further, we reduce the scrub rate to 6 so that SpecIds from the TM hardware will be reclaimed faster. All measures combined deliver the performance shown for *TM opt* in Figure 8.10. For the small input data set (simlarge) and medium thread counts, *TM opt* outperforms the fine-grained locking (cf. Figure 8.10 a)). For the larger input data set (native) the lessons learned are necessary to increase the scalability with TM and come into sight of fine-grained locking (Figure 8.10 b)). The execution with native input data sets increases the input data size and the frame rate simultaneously. We were interested to know which of these parameters influences the performance in favor of TM. The result is that the smaller input data favors TM whereas the frame rate simply serves as a multiplier of the observed performance. Further, this experiment reveals that the BG/Q architecture is extremely stable with very little noise. The observed performance of the synchronization patterns shows that even with Hardware Transactional Memory, expert-level use of lightweight efficient fine-grained locks will be hard to beat. Moreover, we identified the need to research tools with support for TM that enable an in-depth understanding of the TM behavior and the causes for performance degradation. From a programmability perspective, employing TM is as simple as using a single global lock.



(a) Input data set *simlarge*.



(b) Input data set *native*.

Figure 8.10: `fluidanimate` with coarse-grained and fine-grained locking as well as simple and optimized transactions.

For this experiment, even the unoptimized TM version outperforms the single global lock in terms of speedup and scalability. Therefore TM takes an important step towards simplifying shared memory programming.

8.7 Summarizing the First Experience with BG/Q

In this chapter we evaluated BG/Q's TM hardware from the perspective of an application developer. We introduced CLOMP-TM, a benchmark designed to represent scientific applications, and used it to contrast HTM against traditional synchronization primitives, such as *omp atomic* and *omp critical*. We then extended CLOMP-TM with MPI to mimic hybrid MPI/OpenMP parallelization. Additionally, we studied the impact of environment variables on the performance. Finally, we condensed the findings into a set of best practices and applied them to a Monte Carlo Benchmark. An optimized TM version of MCB with 64

threads achieved a speedup of 27.45 over the baseline. Further, an optimized TM version of the Smoothed Particle Hydrodynamics method from the PARSEC suite achieved a speedup of 14.5 with 64 threads and significantly outperformed a simple TM version (speedup of 4.4) as well as a coarse grain lock (speedup below 1) and verified the usefulness of the best practices. Moreover, our results also show that an expert-level use of lightweight efficient fine-grained locks is hard to beat with TM. For MCB, a synchronization pattern that combines omp atomic and omp critical achieved a slightly larger speedup of 27.57 over baseline. These findings illustrate that performance with HTM does not come for free and, even when following the guidelines developed and presented in this chapter, performance with TM may not quite measure up to the expert-level use of locks for the scientific applications considered. However, TM comes with the advantage of improving the programmability and productivity because the user does not have to explicitly manage locks (which is known to be error-prone). Thus, the use of TM or locks depends on the expected gain when comparing development effort with performance improvements. Our findings motivate further research to refine our lessons learned and develop tools for programming with TM.

9. Tool Support for TM on BG/Q

This chapter presents three tools for HTM with implementations for the BG/Q architecture. This idea has also been sketched in [177]. Section 9.1 motivates the need for specific tools that support HTM. The design of the tools, that feature profiling, tracing and measuring overheads of transactions, is presented in Section 9.2. All tools incorporate the reading of architecture-specific hardware performance counters. Section 9.3 illustrates how the tools with these performance counters help to track down performance issues in applications with transactions. In particular, we profile a hydrodynamics proxy application, present the results in Section 9.4 and reveal the cause for suboptimal TM performance. Visualization of the trace data with Vampir is shown in Section 9.5. Section 9.6 briefly compares the approach with related work and Section 9.7 concludes this chapter.

9.1 Introduction and Motivation for Tools on BG/Q

The BG/Q architecture is the first commercially available architecture that supports hardware transactional memory. As we already described in the previous chapter, the user must follow a set of guidelines in order to exploit the full potential that the new synchronization mechanisms of this architecture offer, where each node is capable of running 64 threads in parallel. The guidelines include the synchronization with Transactional Memory (TM) in order to select a synchronization mechanism for the application that yields the best performance. Due to the optimistic concurrency, the conflict probability with other transactions is the dominant factor for the performance of TM. Chapter 8 shows that a low conflict probability alone is not sufficient for TM to be the fastest synchronization alternative. According to these findings, the transaction length also plays an important role for the performance as well as using the correct settings for TM related environment variables. A different study compares the performance of TM in the two execution modi on BG/Q with Software Transactional Memory with privatization and OpenMP critical [201]. This paper provides insights in the architecture of the TM subsystem and gives additional advice when to use which TM mode. For a programmer this kind of guidance is very important, but does not provide the insights needed to understand and tune a particular case. Hence, we identify key performance counter event types that not only explain the observations of the current TM performance, but are also indicative of TM performance with respect to other synchronization mechanisms. We present three tools for TM that combine these

performance counters with statistics from the TM run time to help the programmer to find and correct the cause for the lack of performance. We present a profiling tool for hybrid parallelized programs with MPI and OpenMP that is capable of aggregating TM statistics and performance counters for each MPI rank to obtain an overview over MPI execution with OpenMP and TM. This helps to isolate performance issues in particular ranks. For an in-depth investigation of these ranks, we present two additional tools for TM. The next tool generates traces of the execution of the TM application and preserves the information as snapshots so that the application behavior can be visualized in great detail. The final tool for TM targets overhead measurements that enable us to obtain a better understanding of the utilization of the architectural resources during the execution phases of a transaction.

This chapter makes the following contributions to the state-of-the-art:

- enriches the set of best practices for efficient synchronization with TM with a set of tools that enable each programmer to explore the subtleties of TM execution:
  - the first tool that profiles applications using MPI and OpenMP with TM on BG/Q,
  - a tracing tool for TM that enables in-depth inspection of thread-level execution and utilization of the architecture through visualization with a state-of-the-art visualization tool,
  - a tool that measures overheads associated with TM, designed to dissect these overheads and direct optimization efforts for the TM stack,
- presents detailed multi-threaded overhead measurements of the TM subsystem dividing transactional execution into three separate phases in Section 9.3.2,
- uncovers the subtle interaction of the TM system and the prefetching on BG/Q and study the implications for design of the application and choosing the TM mode,
- obtains a comprehensive understanding of the performance of synchronization mechanisms in *LULESH*, a Lagrange hydrodynamics proxy application, and finds the cause for the missing performance with TM.

9.2 Design of a TM Tool for IBM's Run Time Stack

9.2.1 A Profiling Tool for TM

The requirements for a profiling tool for TM follow those of profiling in general: low-intrusiveness and a low memory footprint that does not increase with a longer program run time. In order to fulfill these requirements, we adopt the *caliper concept* that has already been successfully applied in other tools such as TAU [183]. The user specifies the code regions of interest by wrapping them in a *caliper begin* and *caliper end* primitive. In the following, we will refer to the entity that is defined through the mentioned primitives as a *bucket*. During the execution of the program, all counted values are aggregated in the respective buckets as indicated through the begin and end statements. Thus, a bucket holds the performance counter data of the code that it embraces. In case the code iterates over a bucket, the tool updates its performance counter values each time and aggregates them. Through this profiling approach, the tool's memory consumption is linear in the number of buckets, not in the run time of the program. Hence, the tool design enables a longer

```
1 tmt_begin (char *bucket_label,
2             char *values);
3
4 tmt_end (char *bucket_label,
5           char *values);
6
7 tmt_annotate (void *addr, int MPI_type,
8                 char *var_description,
9                 char *bucket_label);
```

Listing 9.1: Application programming interface for the TM profiling tool.

program run time without increasing the demand for memory. This is an essential property of the tool in order to profile large-scale applications. Naturally, the aggregation step reduces the amount of information so that some details of the application execution will not be available during the post processing step. In order to minimize the loss of information, each bucket records the running minimum, maximum and sum of the performance counter readings so that outliers are preserved.

Although the use of the profiling tool requires manual code changes, these changes are straightforward and enable to direct the profiling efforts to application parts of interest and continously refine the profiling data through adding/removing instrumentation. Listing 9.1 illustrates the application programming interface that the user may utilize to instrument the application. The matching functions `tmt_begin` and `tmt_end` open and close a bucket respectively. These buckets support the nesting of buckets. In this case the outer nesting level will also contain the counters of the inner nesting level. Further, buckets may partially overlap. Then both buckets count the overlapped code region. The first parameter of both API functions is a string that defines the name or label of the bucket. This label is used to match the opening and closing of a bucket. At the opening, the tool allocates memory for the bucket and reads all specified counters and saves these readings in a structure that is maintained per thread for thread-local counters or globally for shared counters. The counters (for performance events and TM statistics) then continue to track the code section that the bucket contains. At the end of the bucket, the call into the tool triggers the reading of the counters to a temporary buffer and compares the bucket labels whether this end call matches the last begin. If this is the case, the tool computes the differences of the respective counters and processes these in the following way: each bucket holds data structures for each of the counters that track the minimum, maximum and sum. The semantic differs slightly between thread-local and shared counters: for the thread-local counters (e.g., L1 hits, TM statistics) the minimum refers to the minimum of all computed differences for this bucket of all threads. The same holds for the maximum but the sum is summarizing over all differences of that bucket and all threads. Thus, an average that falls within the minimum and the maximum requires to divide the sum by the number of threads and the number of times the buckets has been accessed. For the shared counters, the minimum, maximum and sum all apply to invocations of the bucket because only one thread reads these counters. Hence, the average can be computed by dividing the sum by the number of accesses to the bucket.

As a second parameter, the user may pass application-specific data to the tool. This additional mechanism enables the correlation of data from the application domain with performance data. This relation enables to attribute a performance issue, e.g., to a specific time step or application parameter that changes over time. However, this parameter is

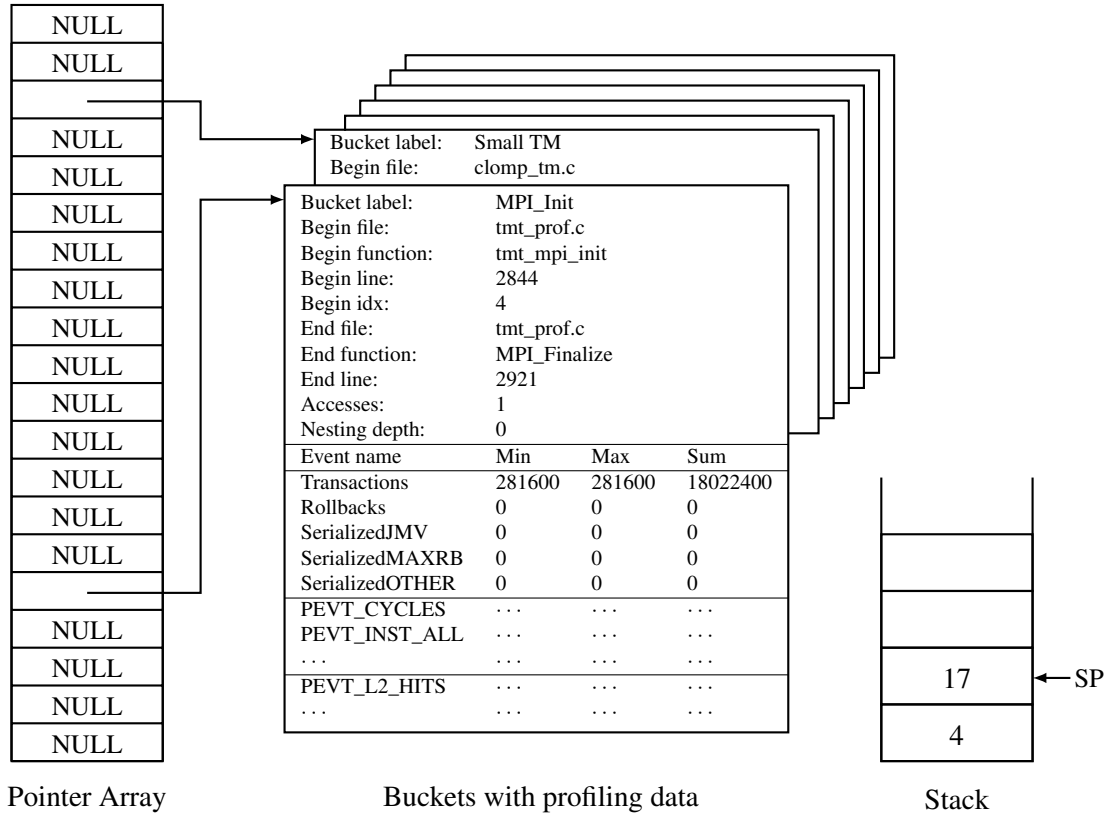


Figure 9.1: Data structures of the profiling tool for TM with a simplified example showing two nested buckets.

more suited for the tracing tool – that is discussed later, but uses the same API – where the user can inspect it during the post-processing step. For the profiling tool the third function in the API, `tmt_annotate`, enables to associate a variable with a bucket. Once this relationship is established, each time the bucket is accessed and read, the variable(s) that are registered with this bucket will also be read. Since the user needs to pass the MPI type when registering the variable, the variable may be subject to the same computation as the regular counter data from the bucket. This means that a running minimum, maximum, and sum of the variable describe the range and the average is computed by dividing the sum through the number of times the bucket has been accessed.

Figure 9.1 illustrates the data structures of the TM profiling tool. A plain pointer array holds references to all buckets. A hash function determines the index of a bucket in the pointer array through hashing over the bucket label, begin file, begin function, begin line and backtrace (if desired). A hash collision transforms the first bucket into a linked list. Thus, from then on this list must be walked to retrieve the correct bucket. At the begin of a bucket, the tool computes the index, looks up the bucket with it or allocates a new one. Further, the tool pushes the index on the stack to keep track of all active buckets. At the end of the bucket, a pop operation on the stack retrieves the last index and the look up of the bucket continues. In addition to the counter data, the buckets with the profiling data also hold the label of the bucket, file, function, number of the begin and the end call, the number of accesses, the nesting depth and a backtrace that includes the active function calls that form the call path for this bucket. This information suffices to identify problematic spots in the execution phases of a program and to correlate them with the source code.

The tool can also detect cases where a problematic execution only manifests itself depend-

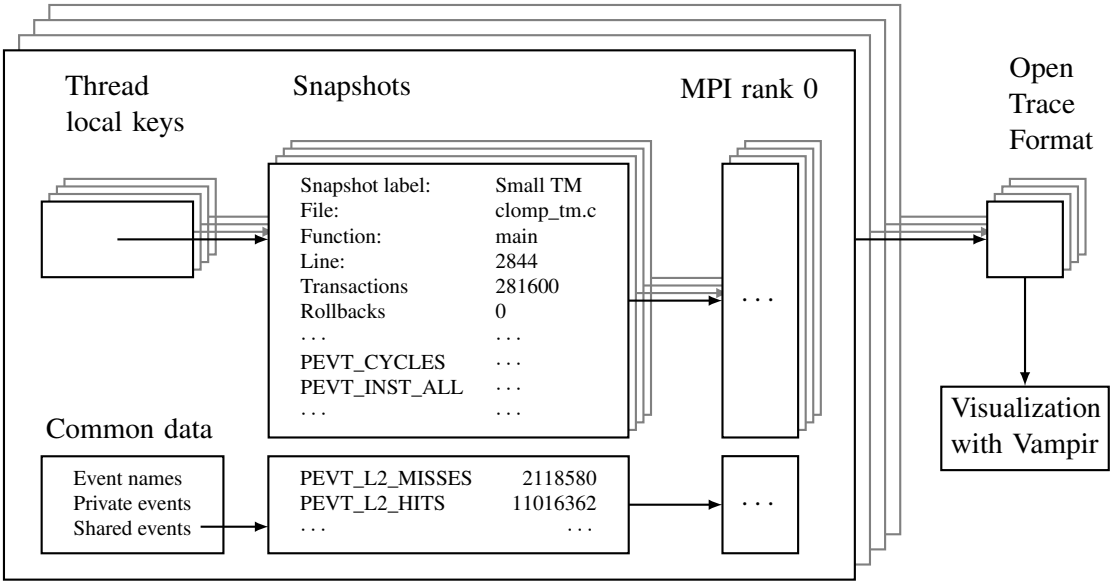


Figure 9.2: Overview of the tracing tool for TM with data structures and work flow.

ing on the call path. The tool provides the *backtrace* option that creates a new bucket in case a program executes the same `tmt_begin` statement through a different call path. The tool implements this option through additionally passing the backtrace to the hash function that determines the index of a bucket. With these additional buckets, the tool reveals the correlation of a problematic application behavior with a specific call path. The additional buckets increase the memory consumption so that it is linear in the number of defined buckets times the maximum number of different call paths of these buckets. In practice the number of different call paths for a bucket is low so that the additional memory consumed is low. These profiling measures are adequate to gain an overview of the (TM) performance of the application, identify imbalanced threads, take counter measures to optimize the program and direct the attachment of a tracing tool.

9.2.2 A Tracing Tool for TM

In order to enable an in-depth inspection of the application's behavior at the thread-level, we also design and implement a tracing tool that complements the profiling tool. The tracing tool uses snapshots of the performance counters and the TM statistics to capture the state of the program execution. In contrast to the state-of-the-art in tracing tools which use asynchronous sampling to achieve a scalable tool that preserves both the calling contexts and time information [198], we rely on the careful placement of instrumentation routines through the programmer. Our methodology focuses the instrumentation to code sections of interest and works well with TM which sampling does not. Asynchronous sampling may interrupt a transaction which will at least change the execution mode of that transaction if not abort it. Thus, the application behavior recorded by sampling a TM region is biased compared with the unobstructed execution.

Figure 9.2 depicts the main data structures of our tracing tool. Thread local keys serve as an entry point to thread specific performance snapshots. Each snapshot contains information on the file, function and line of the program execution as well as a complete TM report obtained from the TM run time system and the counter values of the per-thread BGPM performance counters. The tool saves pointer to the values of the shared BGPM performance counters in a shared data structure that also holds the event names of all

```
1 tmt_snapshot (char *snapshot_label,
2                char *values);
3
4 tmt_annotate (void *addr, int MPI_type,
5                char *var_description,
6                char *snapshot_label);
```

Listing 9.2: API for the tracing tool for TM.

events and the event sets that BGPM requires. Only one thread reads and saves the shared counters at a snapshot. This synchronizes the reading of the private and the shared counters and facilitates to relate both while at the same time being memory efficient. A singly linked list connects the shared counter data as well as the snapshots respectively. In a program with multiple MPI ranks, the tool collects the performance data for all ranks separately, merges the data and writes them to trace files. The tool supports the Open Trace Format (OTF) [113] so that the performance data can be visualized with Vampir [143]. The strength of Vampir is the variety of views, including the performance radar that features an intuitive way of coloring performance data to highlight peaks, that enables a user to compose the performance data in a way that it is most effective for him or her.

This tracing tool differs from the previous tracing tools presented in this thesis in that it does not trace single events, but instead traces snapshots of the program execution. Although the granularity is different compared to the previous tools, the memory requirements are still linear in the run time of the program. Similar to the approach presented in Chapter 5, a fixed buffer for the tracing data that is flushed periodically or when full, could alleviate this issue. Listing 9.2 shows the prototype of the functions that the user inserts in the program code to address the tracing tool. The function `tmt_snapshot` triggers the generation of a snapshot in all OpenMP threads. For more fine grained control a second trigger function exists that allows snapshots on a per thread basis: `tmt_thread_snapshot`. The two function arguments are a label for the snapshot that helps the user to relate it to the code and a string that can be used to pass the contents of an application-specific variable to the tool. `tmt_annotate` serves the same purpose as in the previous tool and registers a variable with a specific snapshot so that the tool saves the content of the variable each time it takes this particular snapshot.

9.2.3 A Tool for Measuring TM Overheads

Previous work on performance analysis of the Transactional Memory subsystem on BG/Q already explored some of the overheads [201]. In the paper, the single-thread performance is evaluated in great detail. L1 instruction per 100 instructions and the increase in instruction path length relative to the serial execution are reported for a single thread and all STAMP benchmarks. These first insights are important and provide valuable information to the reader. In practice, most users are interested to know how the performance and the overheads evolve when running with multiple threads. This area of the TM system of the BG/Q architecture has not been explored exhaustively. In order to close this gap and provide additional performance information to the user, we research how to design a simple tool that is capable of measuring a break down of the overheads associated with transactions. A new tool is required because the TM system of the BG/Q architecture consists of a hardware and a software part. Both parts are closely coupled and, due to the proprietary nature of the architecture, not generally available. Simply attaching a

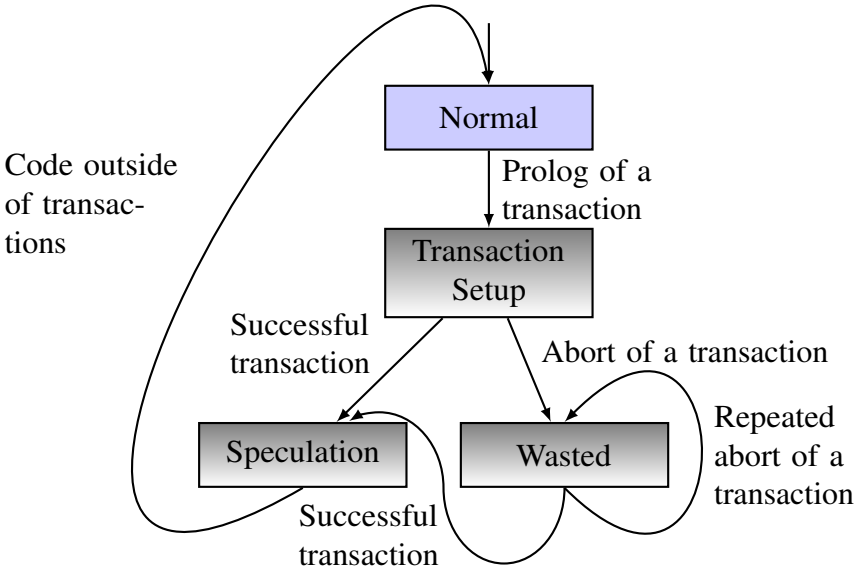


Figure 9.3: Finite state machine that describes how the transactional actions select the correct bucket.

common tool without accounting for the specialties of TM execution, works as long as no speculation takes place. When a transaction executes, the tool could inadvertently interfere with the SW/HW interface, e.g., by overwriting some registers. This can trigger a bug in the OS and cause the transaction to hang. In a second scenario the tool could change the execution mode, e.g., through writing to memory mapped I/O while the transaction is in flight. This would cause the transaction to execute in a special jail mode that serializes the execution. In this case the tool would dramatically change the application behavior and the user would come to a conclusion that would not match the behavior of the application run without the tool. To mostly circumvent these issues, we take an approach that intercepts the function calls into the TM run time at the assembly level. The required information is the function signature of the start and end function of a transaction which we got from the run time group at IBM[1].Through replacing these calls with calls to our tool, we are able to retrieve the performance data. Our tool calls the original function after accumulating the counted values to a specific bucket. A bucket accumulates all counter information that relates to a certain type of execution of the code.

Figure 9.3 illustrates how the transactional actions affect the selection of a bucket. The picture shows four buckets: *Normal*, *Transaction Setup*, *Speculation* and *Wasted*. The three buckets that subdivide the TM execution phases are shown with a gray background. Note that the concept of buckets in the overhead tool is different from that in the profiling tool. The commonality is that both relate execution of code with performance counters. The overhead tool defines implicit global buckets that cover the whole program run and the profiling tool uses explicitly defined buckets that are restricted to the corresponding source code region. The scheme works as follows: prior to setting up a transaction, the tool reads the counters and accumulates them in the *Normal* bucket. The execution before entering a transaction is non-transactional and, thus, corresponds to the *Normal* bucket. Then, during the setup of the transaction, the code prepares the parameters that must be passed into the run time system and saves the minimal context [201]. After that, the actual start of

[1] Amy Wang, IBM, personal communication.

the transaction is initiated by calling into the run time and transferring the control to the hardware. The tool intercepts this call, reads the counters and accumulates them in the bucket *Transaction Setup*. Hence, the bucket *Transaction Setup* contains the preparation of the parameters but not the call into the run time (and the operating system). These are part of the *Speculation* or *Wasted* bucket due to the following technical reason: After the transaction is started through the call into the run time system, our tool also executes under transactional semantics which would undo the collected information during a rollback operation. Moreover, reading or writing performance counters could also influence the execution mode of the transaction and, thus, bias the application behavior. Hence, the careful placement of the calls into our tool prevents these issues because the tool is only called outside of the transactional context. As a consequence, the transaction executes unperturbed and may either be successful, then the tool adds the counters to the *Speculation* bucket, or the transaction aborts, then the tool would add the counters to the *Wasted* bucket. In case a transaction aborts multiple times, all of these executions add to the *Wasted* bucket. Eventually the transaction commits and updates the *Speculation* bucket. After the commit, the non-speculative execution resumes so that the counters add to the *Normal* bucket prior to setting up a transaction. With these 4 buckets the code space is sufficiently divided to distinguish between non-speculative execution, overhead due to setup of transactions, successful speculation and wasted execution due to rollbacks.

9.2.4 Common Implementation Details for the Tools

Integration with MPI

In order to seamlessly integrate the tools with an MPI application, the tools take advantage of PMPI, the MPI standard profiling interface [110]. The MPI standard requires a mechanism to intercept the MPI function calls and defines a duplicate function signature with a PMPI prefix. Often MPI implements these functions as weak symbols that call a `PMPI_` version that implements the functionality or replicate code to avoid extra function call overheads. Thus, our tools need to provide an implementation for all calls it needs to intercept, such as `MPI_Init` and `MPI_Finalize`, to add their own tool functionality that, after performing the profiling or tracing actions, calls the corresponding PMPI function. During the execution of the program, the tool intercepts all MPI calls for which it provides an implementation. So far our tools intercept `MPI_Init` to initialize all shared data structures and setup the tool and `MPI_Finalize` to collect/aggregate the performance data of all ranks and write these to trace file(s).

API and Source Code Information

The API of the tracing and the profiling tool is simple and does not require the user to pass information about the code lines or source files. This information is available to the tool even without the use of debugging symbols. The simple trick uses the macro expansion of the preprocessor. The API functions are defined as macros that expand to function calls with prepended `__FILE__`, `__FUNCTION__` and `__LINE__` macros. The first expansion round of the preprocessor replaces the API calls in the application with these extended function calls. At the end of the expansion process, the preprocessor expands the file, function and line macros so that these confer application level to the tool. Since the preprocessor generates this information, it is independent of the use of debugging symbols. Through exploiting this multi-step expansion process, the user is relieved of supplying the application-level information while the information is still available to the tool.

Interoperability of the Profiling and the Tracing Tool

The API of the profiling tool (cf. to Listing 9.1) shows the same function signatures as the API of the tracing tool (cf. to Listing 9.2). This similarity enables users to instrument the application once for the profiling tool and reuse the same instrumentation with the tracing tool simply through linking to a different tool library. This minimizes the overhead of going from one tool to the other while at the same time preserving the measurement points. However, the semantic of the profiling tool using the bucket concept and the tracing tool with the snapshot concept are different. Hence, the user must be aware of these differences and may also have to arrange for a shorter run time with the tracing tool to account for the higher memory requirements that result in larger trace files. Apart from that, the tracing tool provides an implementation that maps the invocation of `tmt_begin` and `tmt_end` to the `tmt_snapshot` function to ensure the interoperability.

9.3 TM Tools: Experimental Setup and Measurements

9.3.1 Experimental Setup: BG/Q

The experimental setup differs slightly from the one used in the previous chapter and consists of a BG/Q production system with 512 nodes located at the site of LLNL. Unless stated otherwise the experiments run on one node only with 64 threads for the application. The compiler is the IBM XL C/C++ for Blue Gene product in version 12.1.

The Blue Gene/Q compute chip (BQC) is the heart of the node; Haring et al. present a detailed description [82] that we condense in a targeted summary in the following. A BQC consists of 18 cores that surround the on-chip L2 cache. The application may use 16 cores, one is reserved for the operating system and one is a spare that increases the manufacturing yield and is disabled when all cores are functional. The core is based on a PowerPC A2 processor core that is extended by a SIMD quad floating-point unit, a L1 prefetching unit and a wakeup unit. A crossbar connects all processor cores with the L2 cache. Moreover, each core supports 4-way simultaneous multi-threading enabling a total of 64 application threads to run in parallel on one node. The BQC is a system-on-chip that also accommodates two memory controllers, that connect with DDR3 RAM, a network and messaging unit and two interfaces for I/O on the chip.

In the following, we will highlight the A2 processor core with the BQC-specific extensions. The A2 core implements the 64-bit Power instruction set architecture. The 4-way simultaneously threaded core can issue two instructions concurrently: one floating point instruction and one integer, branch, load or store instruction. The execution of these instructions for a thread is in-order. The A2 core has two L1 caches: a 16 KBytes 8-way set associative data cache and a 16 KBytes 4-way set associative instruction cache. Both L1 caches have a cache line size of 64 Bytes. The L1 prefetch unit (L1P) is one of the BQC extensions that supports two types of prefetching: stream and list prefetching. In this work, we only consider stream prefetching that recognizes a stream of contiguous, increasing addresses and prefetches these in chunks of 128 Bytes. L1P holds these in a buffer that accommodates 32 entries and maintains these coherent with other copies in the system. The L1P connects through the crossbar with the L2 cache, the network interface and the PCI-Express interface. The L2 cache is a write-back cache, organized in 16 slices each with 2 MBytes of embedded DRAM and 16-way set associative. In addition, the L2 cache also implements atomic memory operations. The L2 complements these

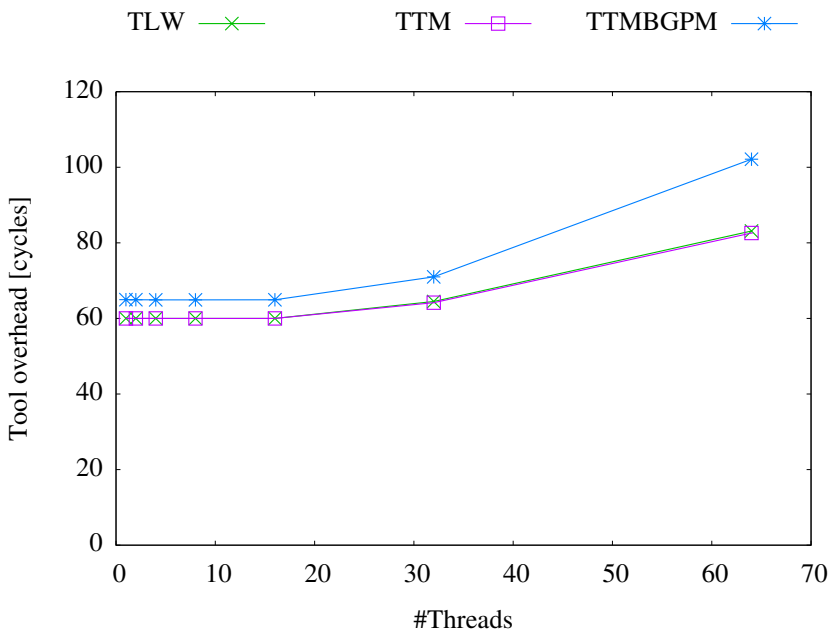


Figure 9.4: Tool overhead for measuring an empty code region with three different levels of information and over multiple thread counts for the TM overhead tool.

with primitives to enable memory speculation required for TM or speculative execution. Speculative changes to memory state are tracked by the L2 and shielded from the main memory. Thus, the L2 also tracks transactions, detects conflicts and supports the commit operation. A specific speculation identifier (short *SpecId*) marks all speculative entries in the L2 of the same transaction. Wang et al. describe the interaction of TM with processor components such as the L1 and the L1P (both denoted in the following as L1) [201]. TM on BG/Q features two different execution modes: the *long-running* and the *short-running* mode. The *long-running* mode uses TLB aliasing to store speculative data in the L1 cache. The L1 cache can accommodate 4 transactional versions (one for each hardware thread) and 1 non-transactional version of a virtual address. This method requires to flush the L1 when entering a transaction because the L2 would not notice e.g., load hits in the L1 otherwise. Through the cache flush, all loads miss in the L1 cache and require a load from L2, which then can track the speculative load. This mode favors long-running transactions that require the L1 because of the frequent reuse of data inside the transaction. The L1 cache flush increases the costs for the reuse of data between code that executes before and inside of the transaction and before and after the transaction. The *short-running* mode, on the other hand, addresses short-running transactions. The mode evicts the line from the L1 on a transactional store. Upon a transactional load the data is loaded from L2, which enables the L2 to track transactional stores and loads after stores. Transactional loads that hit in the L1, notify the L2 via the store queue. Although the *short-running* mode does not flush the cache, this mode comes with a penalty for transactional read-after-write patterns. The programmer may chose the appropriate mode for the application by setting an environment variable before running the program. In the following, we will introduce the methodology to gain in-depth insights into the transactional execution of these modes and the discuss implications on designing the length and contents of transactions.

9.3.2 Tool Overhead of the Overhead Tool

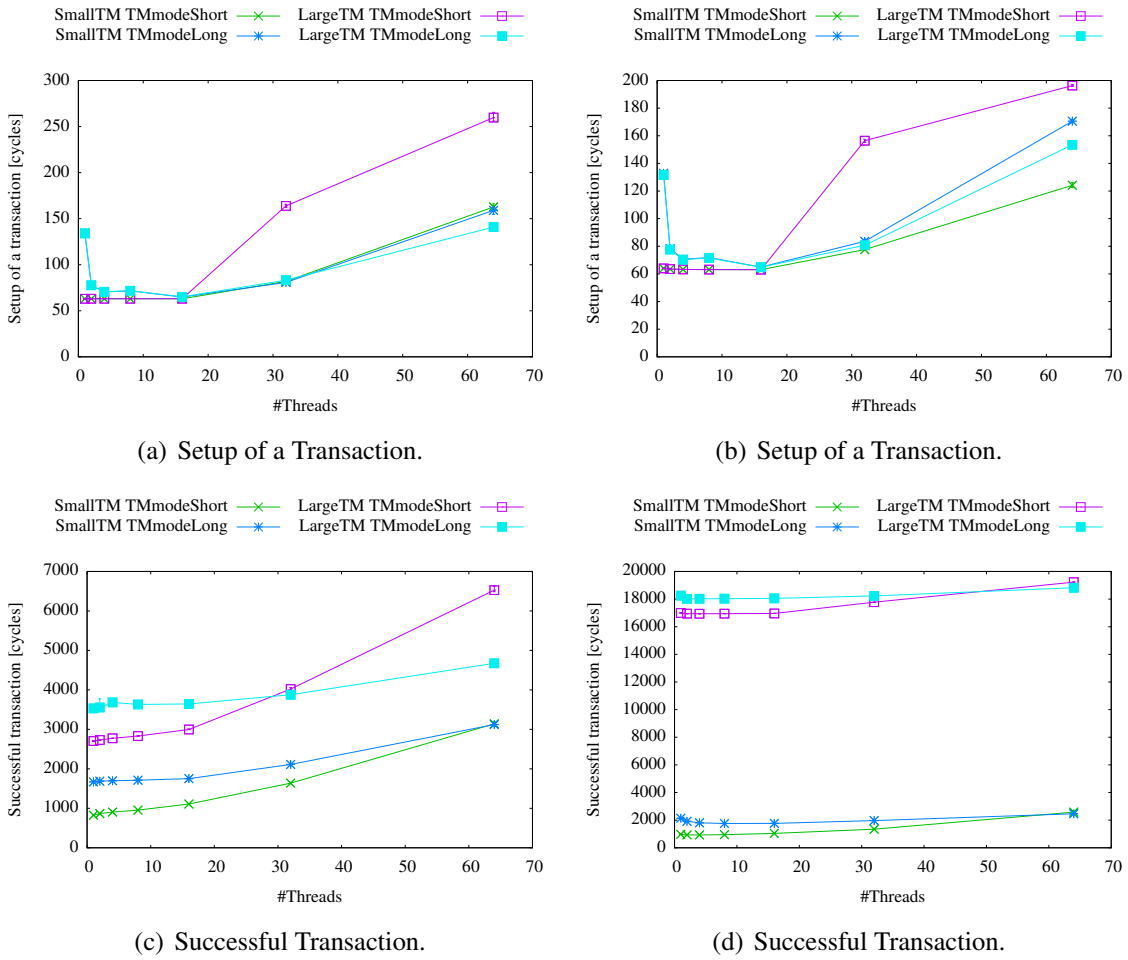
In order to report performance numbers of high quality, we continue with the discussion of the overheads introduced through the tool and the counter measures we take to mitigate the effects. Figure 9.4 depicts the normalized tool overhead for two consecutive accesses to two different buckets. In order to determine the amount of elapsed processor cycles, the tool reads the time base register of the Power PC 64 bit architecture with an inline assembly instruction. The y-axis holds the cycles accumulated in the second bucket normalized to the number of iterations[2]. With this experiment, we estimate the inherent tool overhead that will be subtracted from all reported performance numbers to increase the quality of the reported measurements. The experiment reveals that the number of threads has a slight influence on the tool so that more threads have a higher overhead. Moreover, the amount of information read and processed by the tool also has a slight influence although the counter is read and written first and last in the tool in order to minimize the perturbation of the readings. The *TLW* only uses the cycle counter, *TTM* additionally retrieves TM statistics (at the begin and the end of the run) from the TM run time to use them as summary statistics. Figure 9.4 acknowledges that reading these statistics does not have a large impact on the overhead because it is infrequent and not on the critical path still it may perturb the caches. In case the BGPM counters are also read and reset at the buckets (*TTMBGPM*), we see an increase in the overhead. This overhead with BGPM is expected because the tool requires to read and reset them on the critical path in order to obtain the desired information. Due to reading the time base register at the function entry and exit to the tool, the influence on the reported cycle counts is still low.

9.3.3 Break Down of TM Overheads

In order to precisely understand which part of a transaction requires the most cycles to execute, we conduct an experiment that enables us to correlate the amount of cycles to a specific part of a transaction. For this experiment, we use the tool *TTM* designed for these kind of overhead measurements that Section 9.2.3 describes. The *TTM* tool normalizes the cycles accumulated in the different buckets to the number of executed transactions. Thus, we know how many cycles a transaction spends in each of its execution phases for a given workload. The gathered information will enable us to guide optimization attempts of the current TM system and project execution times of future TM systems. The workload for these measurements is CLOMP-TM with two parameter settings that do not generate contention but differ in the access pattern of the zones and the zone sizes. This variation should increase the reliability of the reported numbers. All reported numbers are averages over 10 runs, the minimum and the maximum of these runs are marked with whiskers in the figures. Due to the low noise in the system, these whiskers are difficult to notice. Gnuplot generated the figures after post-processing with a simple bash script.

Figure 9.5 illustrates the cycles spent in two distinguished phases of a transaction for the long and short-running TM mode on the y-axis over the number of threads. The figure also compares the execution of small and large transactions in CLOMP-TM over the possible range of threads. Figure 9.5(a) reveals that the cycles required for the setup of a transaction differ depending on the length of a transaction. The setup of a transaction comprises the saving of live-in registers. The compiler determines these registers through live range analysis [201]. *Small TM* performs the computation of the extra calculations before the

[2]The test case iterates 10^5 times over the pattern. The result is then normalized to one reading.



(a) Setup of a Transaction.

(b) Setup of a Transaction.

(c) Successful Transaction.

(d) Successful Transaction.

Figure 9.5: Detailed breakdown of a transaction in time for setup and execution distinguishing between execution in long-running or short-running mode across the number of threads. Left hand side run with `clomp_tm_divide1 -1 1 64 100 128 None 3 1 0 6 100` and right hand side with `clomp_tm_divide1 -1 1 256 128 256 stride1,1,stride1%/2 10 1 0 6 100`.

transaction whereas *Large TM* performs the calculation inside of the transaction. Hence, *Large TM* must save more live-in registers which leads to the following observation. On average the short transactions with *Small TM* are faster to set up across all thread counts. For the long transactions with *Large TM*, the setup in the long-running TM mode costs more for low thread counts whereas in the short-running TM mode the higher thread counts are more costly.

Figure 9.5(b) confirms the trends of the first observation although the difference with *Large TM* in short-running mode with 64 threads is not as prominent. For the discussion of Figure 9.5(c) please remember that *Large TM* performs three times (and for the case with the stride even ten times) as many memory updates in a single transaction. Thus, the seemingly higher overheads for *Large TM* are relative to a larger amount of updated memory locations. Taking this into account, *Large TM* with the short-running mode is on a par with *Small TM*. *Large TM* in the long running mode performs better than *Small TM* in the same mode. Especially at higher threads counts, the long-running mode shows a better performance than the short-running mode. For low thread counts, the short-running

mode requires less cycles. Figure 9.5(d) confirms this behavior for the second parameter set with slightly shifted graphs so that the trend is identical.

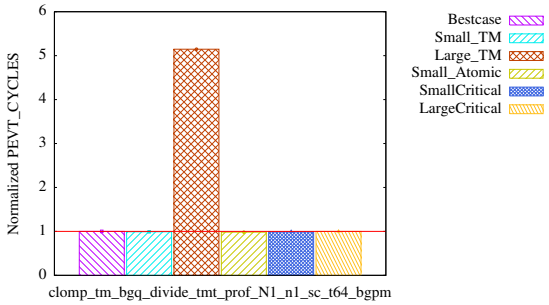
9.3.4 Influence of Scrub Rate on Application's Behavior

In the previous chapter, we demonstrated the influence of the scrub rate on the performance of the CLOMP-TM benchmark. Now, we want to extend the study through identifying a correlation between setting the scrub rate and BGPM performance events with respect to the TM execution mode. Therefore we use the profiling tool, post-process the data with a bash script that uses gnuplot to viusalize the performance data. The resulting bar plots show – in some cases normalized – BGPM event counts or cycles on the y-axis and the buckets on the x-axis.

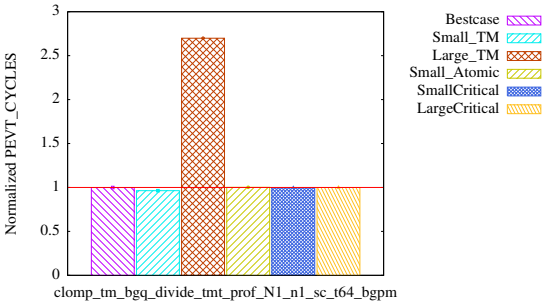
In the following, we present BGPM events measured during an execution with a scrub rate of 66 and a scrub rate 6. In order to clarify the results, we normalize the reported values. Hence, Figure 9.6 shows the ratio of BGPM events of an execution with a scrub rate of 66 divided through the respective event counts with a scrub rate of 6 on the y-axis. The x-axis holds the different synchronization mechanisms. The left hand side shows the execution with the long-running TM mode and the right hand side the short-running TM mode. This figure groups metrics that show similar trends despite the different execution modes. Figure 9.6(a) and Figure 9.6(b) depict that for both execution modes only *Large TM* yields a significant speedup with a scrub rate of 6 while all other synchronization primitives are not affected. Figure 9.6(c) and Figure 9.6(d) show that *Large TM* with a scrub rate of 66 executes between $30\,\%$ and $40\,\%$ more instructions depending on the TM mode. Figure 9.6(e) and Figure 9.6(f) reveal that a scrub rate of 66 fetches at least 3.8 times more instructions for *Large TM* with both TM modes. Figure 9.6(g) and Figure 9.6(h) demonstrate that *Large TM* with a scrub rate of 66 may execute 3.5 times more L1P loads. Further results (without figure) show that the L1P store operations are not affected by changing the scrub rate.

Figure 9.7 illustrates event types where the influence of the scrub rate also depends on the TM mode and shows the influence on the branches executed. Figure 9.7(a) demonstrates that a scrub rate of 66 yields more than 3 times more L1P hits for *Large TM* in the long running mode whereas Figure 9.7(b) shows that the short-running mode yields only ≈ 1.1 times more L1P hits. Figure 9.7(c) illustrates that the changing the scrub rate only has a small impact on the L1P misses with *Large TM* and a scrub rate of 66 may even yields less L1P misses with *Small Atomic*. Figure 9.7(d) highlights that in the short running mode *Large TM* with a scrub rate of 66 yields a lower number of L1P misses. Figure 9.7(e) and Figure 9.7(f) highlight that the number of conditional branches in both TM modes is as least 2 times higher with a scrub rate of 66 and *Large TM*. A similar observation holds for unconditional branches that increase by ≈ 9 times for both modes as illustrated in Figure 9.7(g) and Figure 9.7(h) with *Large TM* and a scrub rate of 66.

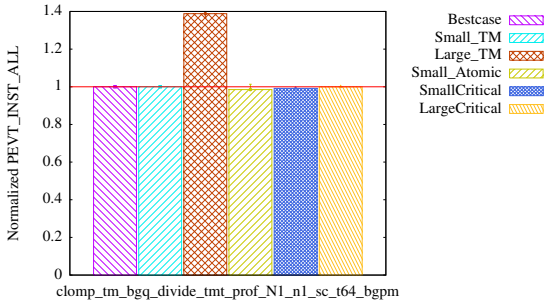
The findings documented in this paragraph are important to understand the various ways in which the setting of the scrub rate may influence the observed BGPM performance events. The identified relationships may help programmers understand the readings of the performance counters better in cases where setting the scrub rate to reduced values is not indicated due to the fact that memory bandwidth, that is used for scrubbing is needed by the application. Moreover, we learned that neither a higher number of L1P hits nor a reduced number of L1P misses automatically leads to better performance. Instead the scrub rate of 6 yields the better performance for both execution modes. This illustrates,



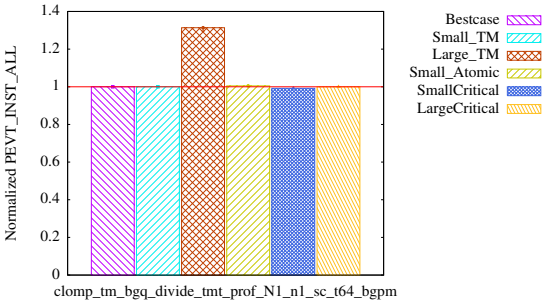
(a) Cycles normalized.



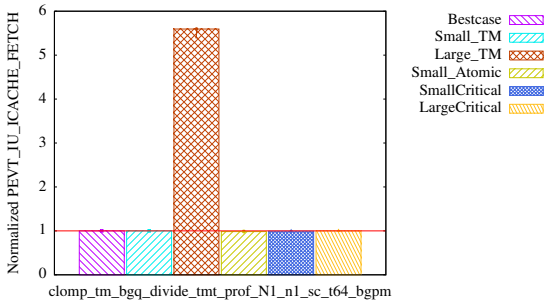
(b) Cycles normalized.



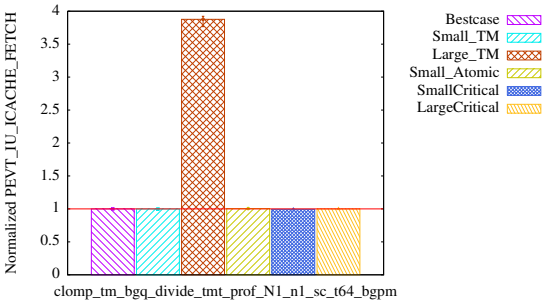
(c) Instructions normalized.



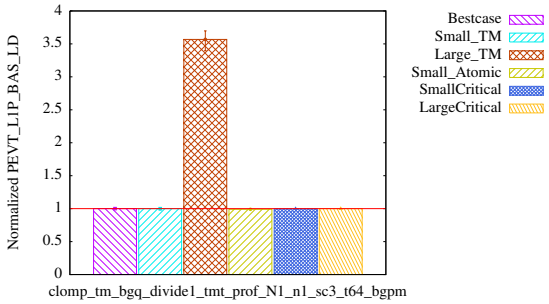
(d) Instructions normalized.



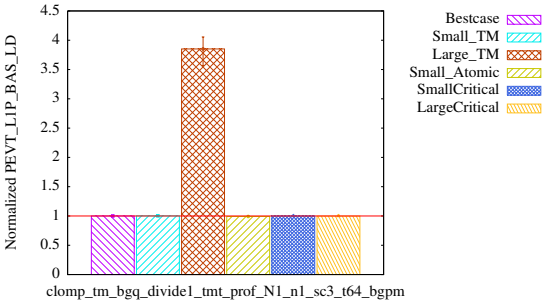
(e) Instruction fetches normalized.



(f) Instruction fetches normalized.

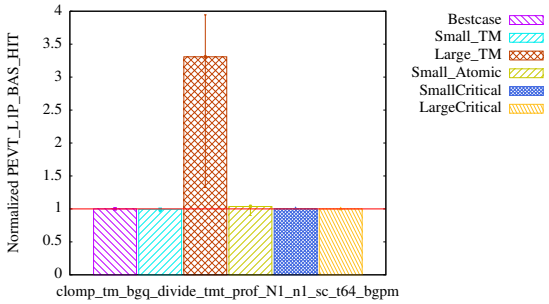


(g) L1P loads normalized.

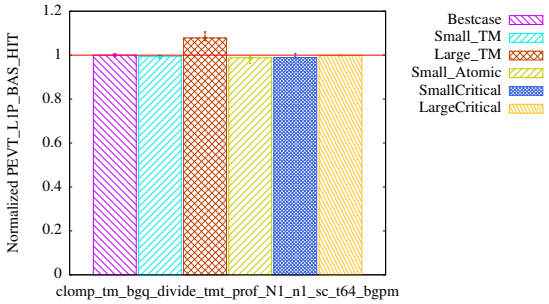


(h) L1P loads normalized.

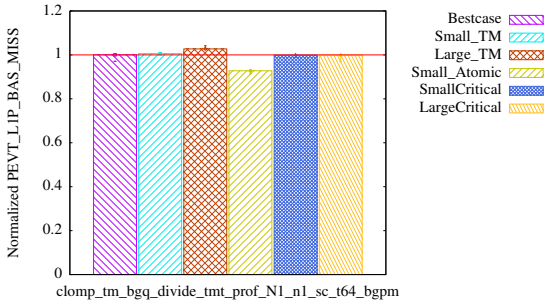
Figure 9.6: Studying the influence of setting the scrub rate from 66 to 6 on transactions in the long-running mode (left hand side) and short-running mode (right hand side). Run with `clomp_tm_divide1 -1 1 64 100 128 None 3 1 0 6 100`.



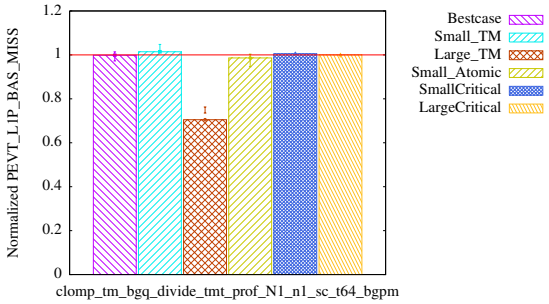
(a) L1P hits normalized.



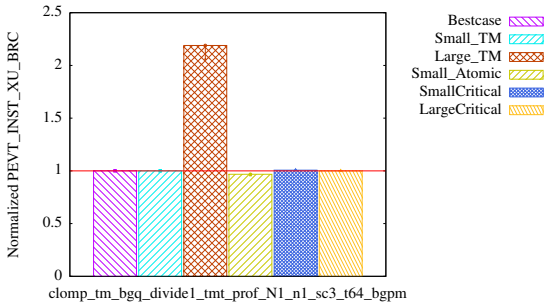
(b) L1P hits normalized.



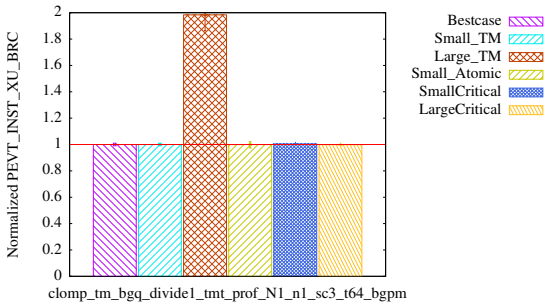
(c) L1P misses normalized.



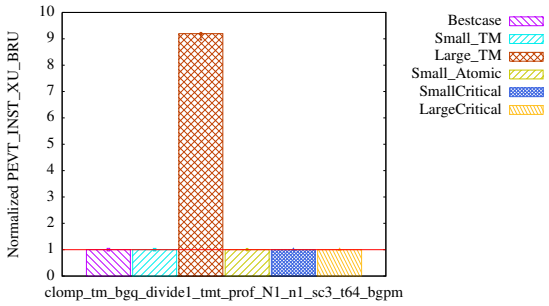
(d) L1P misses normalized.



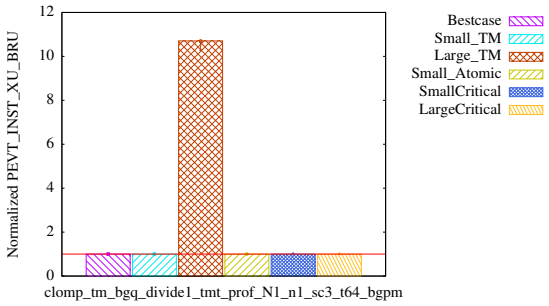
(e) Conditional branches normalized.



(f) Conditional branches normalized.

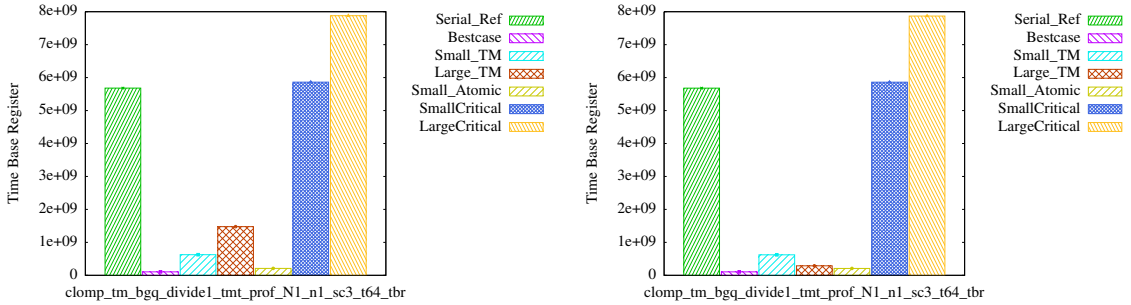


(g) Unconditional branches normalized.



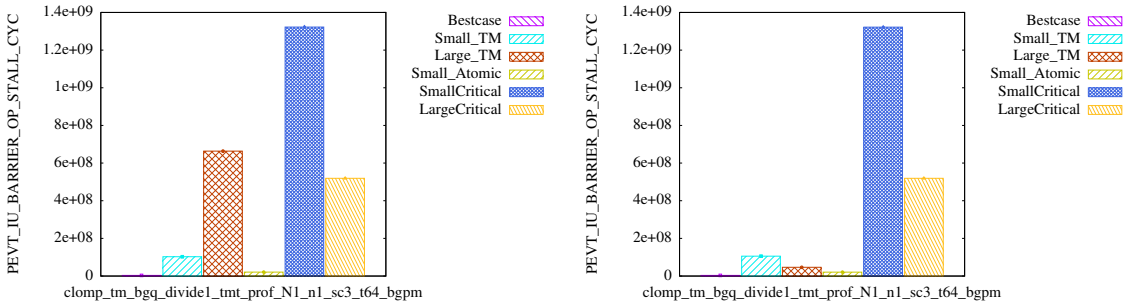
(h) Unconditional branches normalized.

Figure 9.7: Studying the influence of setting the scrub rate from 66 to 6 on transactions in the long-running mode (left hand side) and short-running mode (right hand side). Run with `clomp_tm_divide1 -1 1 64 100 128 None 3 1 0 6 100`.



(a) Cycles with a scrub rate of 66.

(b) Cycles with a scrub rate of 6.



(c) Stall cycles with a scrub rate of 66.

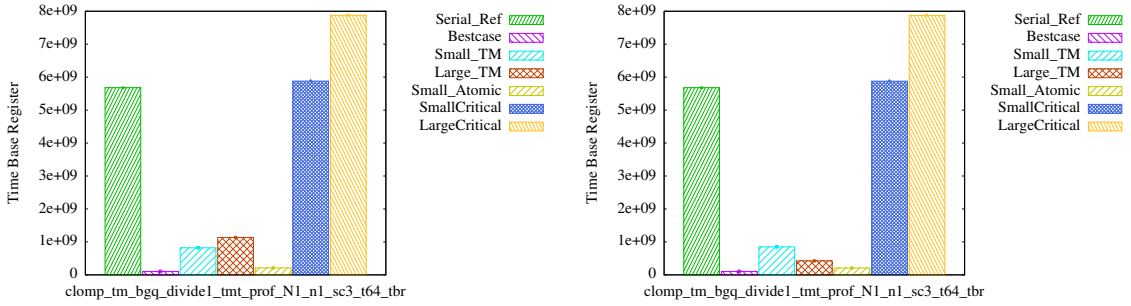
(d) Stall cycles with a scrub rate of 6.

Figure 9.8: Stall cycles as an indicator for adjusting the scrub rate in the long-running mode. Run with `clomp_tm_divide1 64 1 64 100 128 None 3 1 0 sr 100`.

that isolated performance events are not a useful metric to rate the overall performance of the TM application. Instead using a metric that puts these counts in relation to the number of L1P loads and stores would have revealed that the higher number of hits is with respect to an even higher number of loads. From this perspective a user would not set the scrub rate to 66. Moreover, none of the before-mentioned events indicates that the scrub rate benefits from an adjustment. The key event for this is discussed in the following.

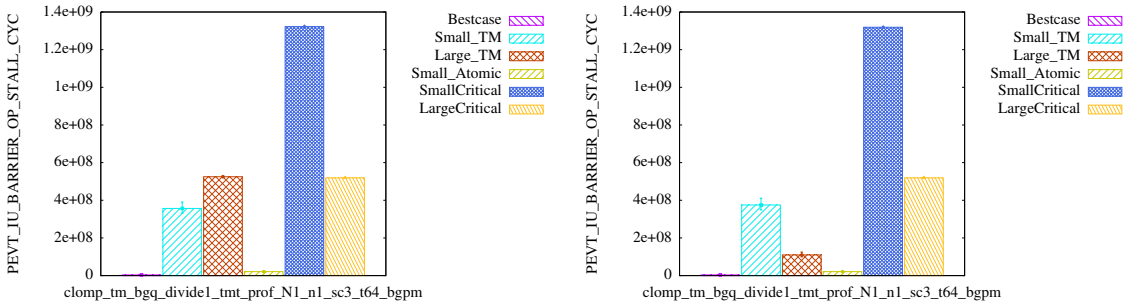
For CLOMP-TM we found a strong correlation between adjusting the scrub rate and the reduced number of stall cycles for *Large TM* represented through the BGPM event PEVT_IU_BARRIER_OP_STALL_CYC. Figure 9.8(a) shows the cycles of the respective synchronization variants with a scrub rate of 66 and Figure 9.8(c) illustrates the corresponding stall cycles. For *Large TM* the number of stall cycles is significant in the long-running mode and decreases when setting the scrub rate to 6. Figure 9.8(b) shows the cycles of all synchronization variants and Figure 9.8(d) illustrates the corresponding stall cycles. The reduction in the overall cycles executed by *Large TM* correlates with the reduced number of stall cycles. *Large Critical* shows the same behavior as before.

For the short-running mode, Figure 9.9 exhibits the same behavior for all synchronization mechanisms. The only synchronization mechanism with a number of significantly reduced stall cycles is *Large TM*. All other synchronization mechanisms (surprisingly including *Small TM*) are not affected. From these observations we conclude that a high stall cycle count for long transactions is reduced through setting the scrub rate to a smaller value, since this allows the hardare to reclaim the *SpecIds* faster.



(a) Cycles with a scrub rate of 66.

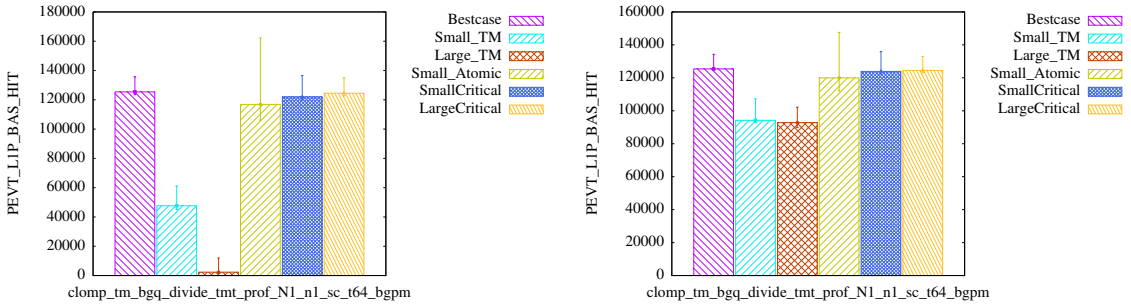
(b) Cycles with a scrub rate of 6.



(c) Stall cycles with a scrub rate of 66.

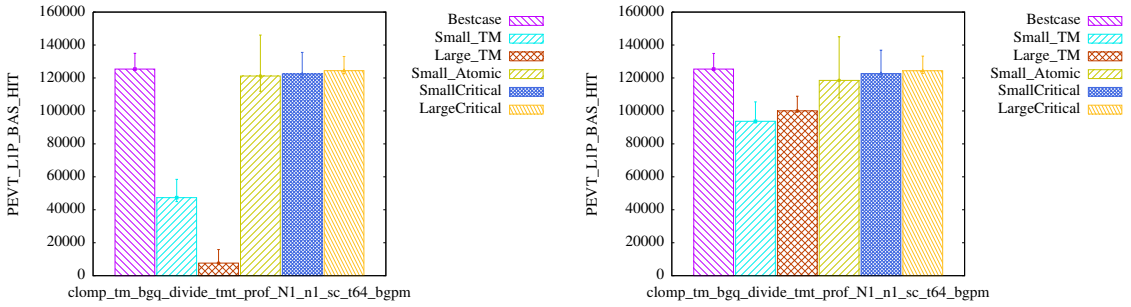
(d) Stall cycles with a scrub rate of 6.

Figure 9.9: Stall cycles as an indicator for adjusting the scrub rate in the short-running mode. Run with `clomp_tm_divide1 64 1 64 100 128 None 3 1 0 sr 100`.



(a) L1P_BAS_HIT in long-running mode with a scrub rate of 6.

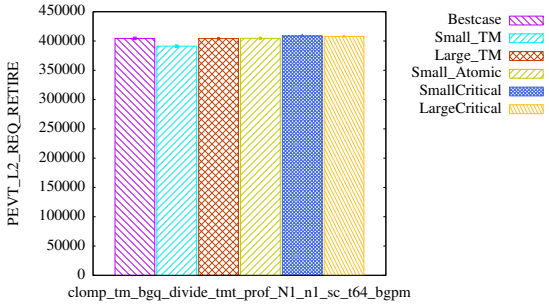
(b) L1P_BAS_HIT in short-running mode with a scrub rate of 6.



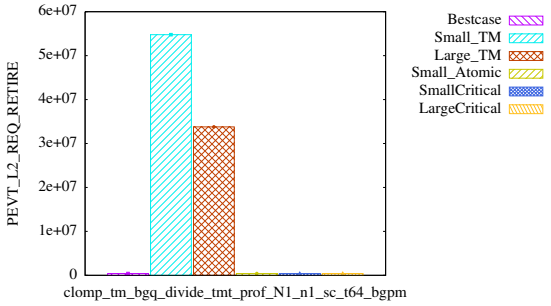
(c) L1P_BAS_HIT in long-running mode with a scrub rate of 66.

(d) L1P_BAS_HIT in short-running mode with a scrub rate of 66.

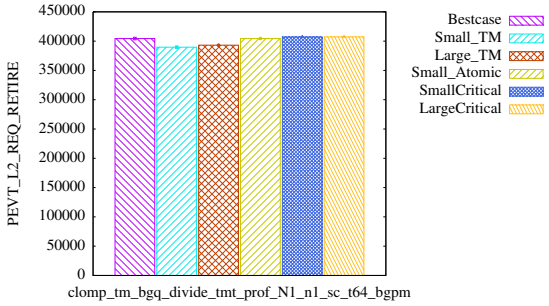
Figure 9.10: L1P utilization in the long-running and short-running mode. Run with `clomp_tm_divide1 64 1 64 100 128 None 3 1 0 6 100`.



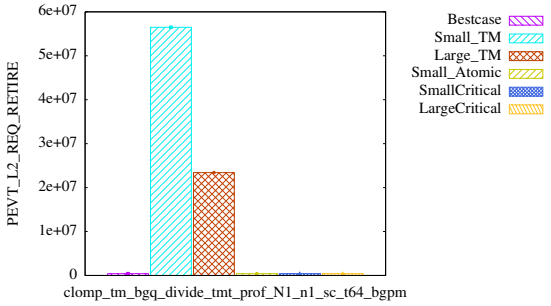
(a) L2_REQ_RETIRE in long-running mode with a scrub rate of 6.



(b) L2_REQ_RETIRE in short-running mode with a scrub rate of 6.



(c) L2_REQ_RETIRE in long-running mode with a scrub rate of 66.



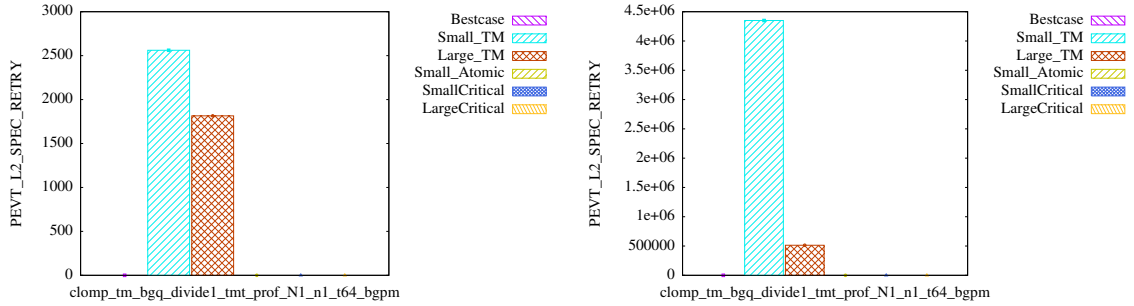
(d) L2_REQ_RETIRE in short-running mode with a scrub rate of 66.

Figure 9.11: L2_REQ_RETIRE and the long-running and short-running mode. Run with `clomp_tm_divide1 64 1 64 100 128 None 3 1 0 sr 100`.

9.3.5 Implications of the TM Mode on the Microarchitecture

Figure 9.10 illustrates the impact of the TM mode on the L1P unit. In the long-running mode, a transaction flushes the L1 data cache before the begin. Hence, all data must be retrieved from the L2 cache. Figure 9.10(a) depicts that both TM variants show significantly less L1P hits than all other synchronization variants with a scrub rate of 6 and the long-running mode. Moreover, Figure 9.10(c) confirms that this observations holds even for a scrub rate of 66. Figure 9.10(b) demonstrates that the hits in L1P are higher for the short-running mode. This also holds with a scrub rate of 66 (cf. to Figure 9.10(d)). The reason is that the short-running mode does not flush the cache; instead it evicts dirty lines from the L1 and propagates loads over the L1P store queue to the L2.

Figure 9.11 demonstrates how the L2_REQ_RETIRE event depends on the execution mode. L2_REQ_RETIRE counts the number of accesses to the L2 that retire after a look up and do not enter the hit or miss queue. Among these events is the L1 hit notification that notifies the L2 of L1 read hits in a transaction in the short-running mode. Figure 9.11(b) illustrates the distribution of the L2_REQ_RETIRE events over the synchronization primitives in the short-running mode and a scrub rate of 6. *Small TM* has the highest counts followed by *Large TM*. In the long-running mode, as illustrated in Figure 9.11(a), all synchronization variants have roughly the same number of events. This does not change when the scrub rate changes to 66 as shown in Figure 9.11(c). For the short-running mode it does have an impact, as can be seen in Figure 9.11(d). The counts for *Small TM* rise slightly while the ones for *Large TM* drop.



(a) SpecRetry event with None and a scrub rate of 6. (b) SpecRetry event with firstParts and same scrub rate.

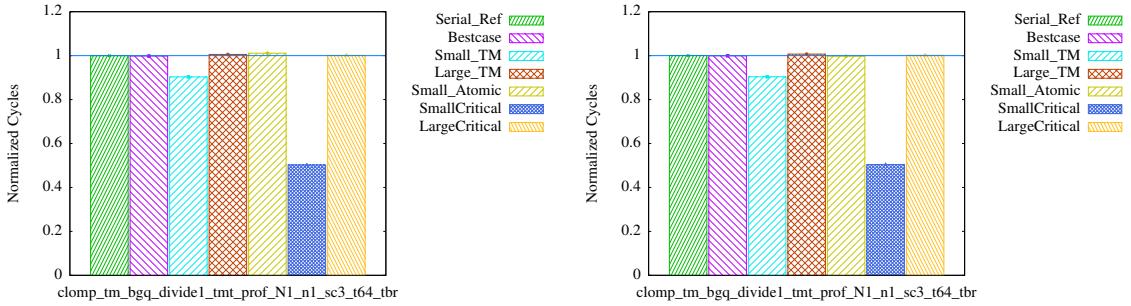
Figure 9.12: Studying the influence of the access pattern on the L2 SpecRetry event with the long-running mode. Run with `clomp_tm_divide1 -1 1 64 100 128 <pattern> 3 1 0 6 100`.

SpecRetry is the event associated with a request made to the L2 that has to be retried later because it conflicts with an ongoing commit operation. The counts of SpecRetry increases by a factor of ≈ 4 for scatter of 2 and *Small TM* when going from stride of 1 to 4 (compared with a factor of ≈ 2 for *Large TM*). We observe the same trend for scatter with 20 memory updates. Our explanation is that the stride of 4 changes the timing of L2 accesses and, thus more operations collide with ongoing commit operations. Although, a factor of 2 to 4 looks large at first sight, the numbers of the SpecRetry event must be viewed relative to the absolute number of updates in a transaction. Then, the relevance of these counts shows that the quotient of $\frac{\#SpecRetry}{\#Transactions} << 1$. Thus, for the two cases with the strides these SpecRetry counts are not relevant. Figure 9.12 illustrate a wider range of measured SpecRetry events. For the case None without contention, Figure 9.12(a) illustrates that the SpecRetry event occurs more often with *Small TM* than with *Large TM*. Figure 9.12(b) demonstrates that the gap for the SpecRetry between *Small TM* and *Large TM* widens with contention (firstParts). Further, the occurrence of the event has a different quality.

Further, experiments (not shown) indicate that changing the scrub rate has an effect on the occurrence of the SpecRetry events but the outcome is difficult to predict. Rather the SpecRetry events and the scrub rate must be considered in relation to other events such as the PEVT_IU_BARRIER_OP_STALL_CYCLES to obtain a better understanding of the behavior.

9.3.6 Long Transactions at Any Cost?

The long-running mode of the TM system requires to flush the L1 cache prior to entering a transaction. In some cases (e.g., *bayes* from the STAMP suite reported in [201]) this leads to a performance that is outperformed by an STM that supports privatization. Let's assume a simple programming pattern where a transaction updates shared variables with the result of a previous computation. Considering the cache flush at the begin of a transaction, a programmer is tempted to avoid the associated penalty through moving the computation inside the transaction. Then, the cache flush would not affect the computed result and provided that the data that serves as input to the computation misses in the L1, this may be a worthwhile scenario. In the following, we will present the observations made by studying the implications of this optimization on the utilization of the microarchitecture. For this experiment, we compare two versions of the CLOMP-TM benchmark. The first version, called *CompBeforeTM* performs the computation for *Small TM* and *Small*



(a) Speedup of *CompInTM* over *CompBeforeTM* with long-running mode.

(b) Speedup of *CompInTM* over *CompBeforeTM* with short-running mode.

Figure 9.13: Studying the influence of moving the computation inside the transaction with both TM modes. Run with `clomp_tm_divide1 -1 1 64 100 128 None 3 1 0 6 100`.

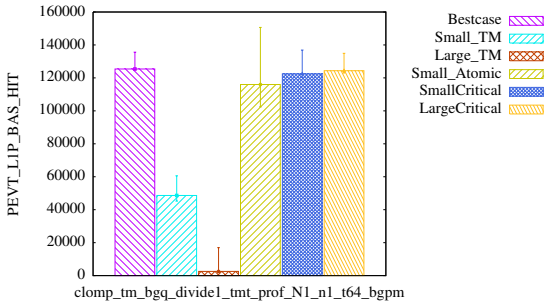
Critical before the transaction or critical section respectively. The second version, called *CompInTM*, moves these computations inside the transaction/critical section. At a first sight, this code change conforms with the policy of favoring longer transactions. Note that these changes do not affect *Large TM* because the large transactions already include the computation. Figure 9.13 illustrates the speedup of *CompInTM* over *CompBeforeTM*. Figure 9.13(a) demonstrates that this simple change does have a performance impact on *Small TM* in the long-running mode, reducing its performance by $\approx 10\,\%$. For *Small Critical* the performance reduces by $\approx 50\,\%$. In the short-running mode, as Figure 9.13(b) illustrates, the picture is identical. Regardless of the TM execution mode, *CompInTM* shows a slowdown compared with *CompBeforeTM*. For *Small Critical* the performance penalty is even higher than for *Small TM*.

For *Small Critical* the longer critical section size in the *CompInTM* version yields longer hold times of the lock that in turn blocks other threads, which spin on the lock acquisition. As a side effect more instructions are executed in total. For *Small TM* this simple explanation does not hold as transactions are lock-free. Thus, each transaction may proceed as soon as it starts. A closer look at Figure 9.14(a), which shows the L1P hits of *CompBeforeTM* and Figure 9.14(b), which shows the L1P hits of *CompInTM* reveals that the already low amount of the L1P hits degraded further. Figure 9.14(c) and Figure 9.14(d) complement the picture by showing the L1P misses. Thus, the simple change significantly reduced the number of L1P hits for *Small TM*. Since the computation executes a stride that touches all cache lines of a zone and is prefetchable, the transaction prologue not only flushes the L1 caches but also perturbs the prefetching in a way that affects TM performance.

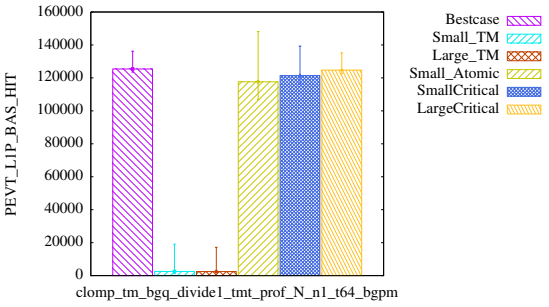
Further, Figure 9.15 uncovers that the stall cycles increase by $47\,\%$ for *CompInTM* compared with *CompBeforeTM*. This fits the picture that *CompInTM* yields higher latencies for memory accesses because the frequent misses in the L1P.

9.4 Profiling LULESH

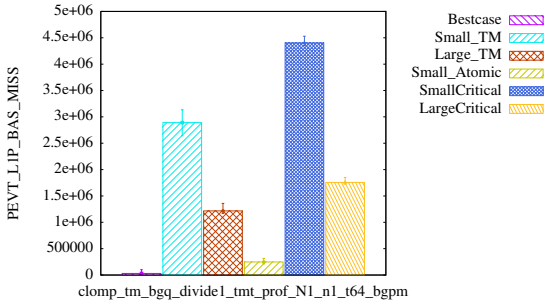
LULESH stands for Livermore Unstructured Lagrange Explicit Shock Hydrodynamics application that solves the Sedov blast wave problem for a symmetrical blast wave in three dimensions modeling one octant. LULESH is one of the challenge problems in the DARPA Ubiquitous High Performance Computing program. The idea is to test optimization



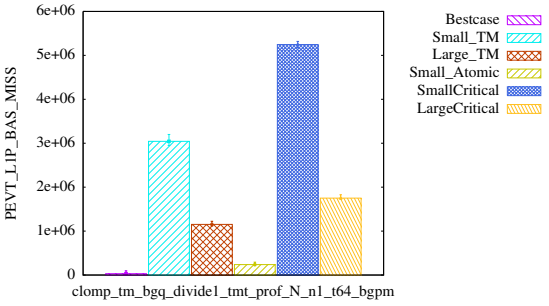
(a) Hits in L1P with *CompBeforeTM*.



(b) Hits in L1P with *CompInTM*.

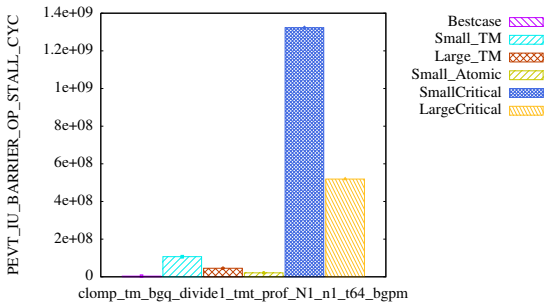


(c) L1P misses *CompBeforeTM*.

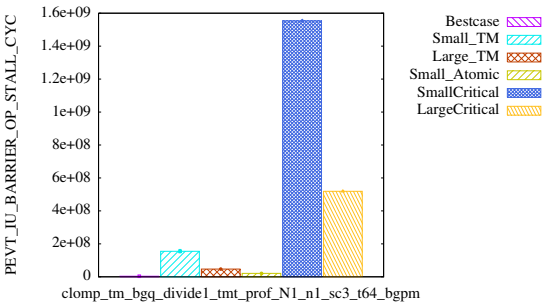


(d) L1P misses *CompInTM*.

Figure 9.14: Studying the influence of moving the computation inside the transaction in the long-running mode. Run with `clomp_tm_divide1 -1 1 64 100 128 None 3 1 0 6 100`.



(a) Stall cycles with *CompBeforeTM*.



(b) Stall cycles with *CompInTM*.

Figure 9.15: Studying the influence of setting the scrub rate on the stall cycles with the long-running mode. Run with `clomp_tm_divide1 64 1 64 100 128 None 3 1 0 6 100`.

strategies on LULESH and if they are successful port them back to production codes. LULESH uses a Lagrangian hydrodynamics methodology. In contrast to the Eulerian formulation that expresses flow variables at fixed spatial positions over time, the Lagrangian formulation defines flow variables as functions of time and material elements [120]. While the Eulerian formulation leads to material flowing through a fixed mesh, the Lagrangian formulation constructs the mesh of material elements and aligns their interfaces with the element boundaries. Over time the mesh changes according to the movement of the elements. Lagrangian methods are suited to model multiple materials or moving boundaries. A Technical Report holds all details of LULESH and the underlying physics [120] from which we extract the steps of the algorithm. LULESH first creates the domain(s), then the material index set, initializes the problem state and creates the boundary conditions. Then LULESH carries out the following two steps until the final simulation time is reached: calculate the time for the next increment and advance the variables by the calculated time using the leap frog time integration method. The leap frog scheme first computes the new values for the node variables and then for the element variables.

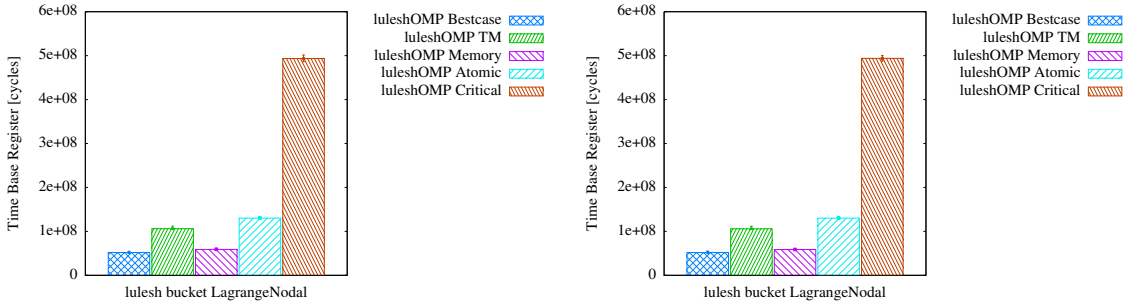
In the following, we demonstrate the usefulness of the profiling tool for TM in order to find the best synchronization mechanism for LULESH. First, we loosely divide the LULESH application by augmenting it with calls to the buckets. We introduce a *TimedBucket* that embraces the timed loop. Inside the timed loop, we add the *LagrangeLeapFrog* bucket and the *TimeStepIncrement* bucket on the next nesting level. The *LagrangeLeapFrog* bucket is further subdivided into three small buckets one of them being the bucket *LagrangeNodal* on the inner most nesting level. These five buckets partition the application in sufficiently small parts so that a thorough analysis of the application is feasible without too high overheads in run time.

We run the LULESH version that uses OpenMP threads, short *luleshOMP*, with the following synchronization mechanisms: Bestcase, TM, Memory, Atomic, and Critical. Note that TM, Atomic, and Critical only differ in the synchronization mechanisms used. Bestcase does not use synchronization (and potentially reports a wrong answer). Memory avoids the use of synchronization primitives through the use of additional memory. A first look at the measurements of luleshOMP TM reveals that only *LagrangeNodal* executes transactions. Thus, in the following we will focus the discussion on the bucket *LagrangeNodal*. Please note that the cycles reported here are per thread and per execution of the embracing code section. Thus, to assess the impact on the overall run, the reported values must be multiplied by the number of accesses to the buckets, which is 1816 times in case of the bucket *LagrangeNodal* for the chosen input data set[3].

LULESH provides interesting insights in the interaction of the scrub rate with the chosen mode. The long-running mode generates almost identical results when the scrub rate goes from 66 to 6. Whereas the short-running mode responds with a speedup of 1.96 x.

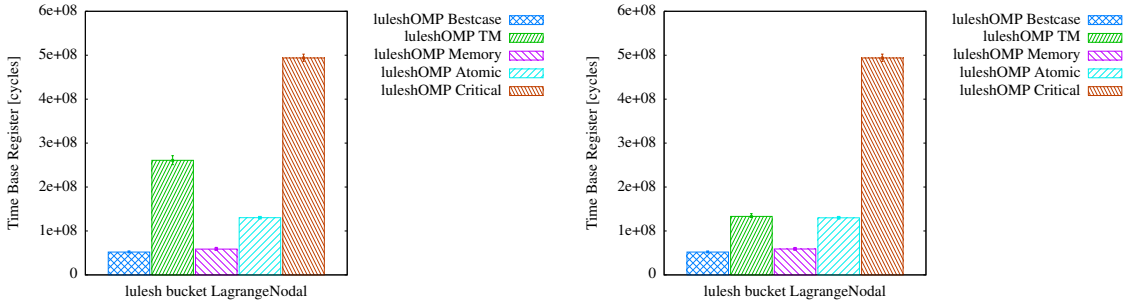
Figure 9.16 shows the results from running LULESH in the long-running TM mode with respect to other synchronization mechanisms by showing the number of cycles for the execution of the *LagrangeNodal* function on the y-axis and the different synchronization variants on the x-axis. The scrub rate is set to 66 for Figure 9.16(a) and to 6 for Figure 9.16(b). Because adjusting the scrub rate may also influence the other synchronization mechanisms through reducing the available memory bandwidth. Both figures are almost identical, which means that in the long-running TM mode reducing the scrub rate for

[3] All LULESH runs solve a problem with 70^3 elements.



(a) Scrub rate of 66. (b) Scrub rate of 6.

Figure 9.16: LULESH executed with different scrub rates and long-running TM mode.



(a) Scrub rate of 66. (b) Scrub rate of 6.

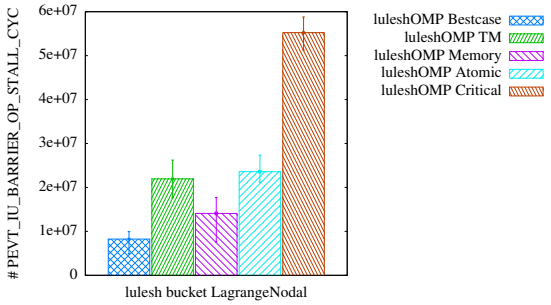
Figure 9.17: LULESH executed with different scrub rates and short-running TM mode.

LULESH neither has an effect on TM performance nor affects the execution of the other synchronization mechanisms.

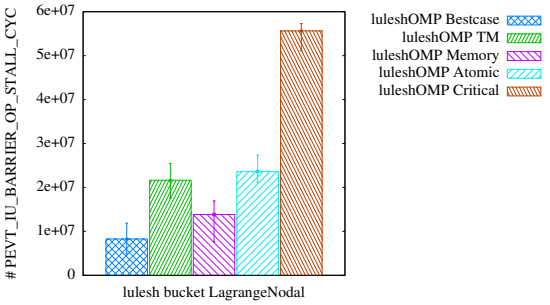
Figure 9.17 reports the results from running LULESH in the short-running TM mode. In this case adjusting the scrub rate from 66 (cf. to Figure 9.17(a)) to 6 (cf. to Figure 9.17(b)) yields a speedup of 1.96 for the execution with TM. Again, the other synchronization variants are not affected by tuning the scrub rate.

Figure 9.18 reveals the reason for the low performance with *TM* in the long-running mode: misses in the L1P unit. First, the stall cycles for TM are high compared with *Bestcase* but reducing the scrub rate does not have a positive influence as shows a comparison of Figure 9.18(a) with Figure 9.18(b). The L1P hits of *TM* are even higher than those of *Bestcase* and constant under changes of the scrub rate as is illustrated in Figure 9.18(c) and Figure 9.18(d). Only the L1P misses that are shown in Figure 9.18(e) and Figure 9.18(f) are the highest for *TM*.

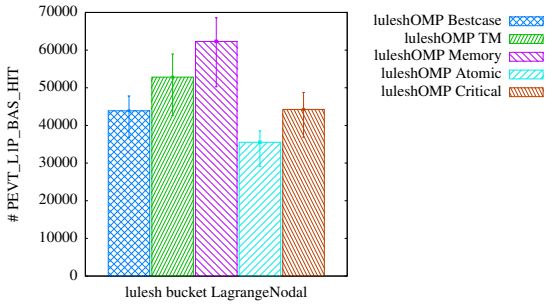
Figure 9.19 reveals the reason for the low performance with *TM* in the short-running mode: misses in the L1P unit, which is the same reason as with the long-running TM mode. First, the stall cycles for TM are the highest compared with the rest but reducing the scrub rate has a positive influence as the comparison of Figure 9.19(a) with Figure 9.19(b) reveals. The L1P hits of *TM* are not as high as those of *Bestcase* and constant when changing the scrub rate as is illustrated in Figure 9.19(c) and Figure 9.19(d). Although for the short-running mode the L1P misses that are shown in Figure 9.19(e) and Figure 9.19(f) are the highest for *TM*. Hence, we found that the same reason causes the low performance for both TM modes compared with *Bestcase* or *Memory* and 64 threads.



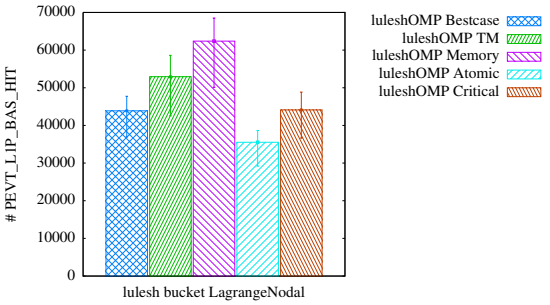
(a) Stall cycles with a scrub rate of 66.



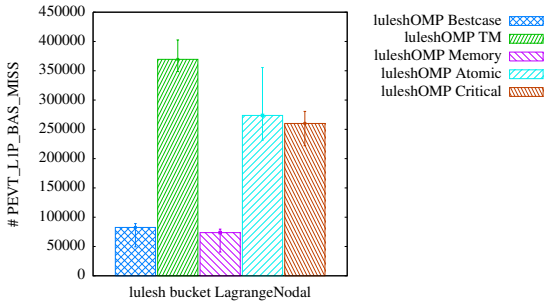
(b) Stall cycles with a scrub rate of 6.



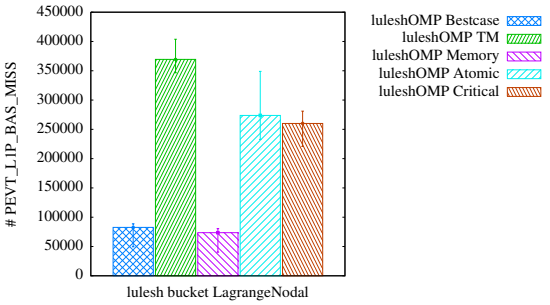
(c) L1P hits with a scrub rate of 66.



(d) L1P hits with a scrub rate of 6.



(e) L1P misses with a scrub rate of 66.

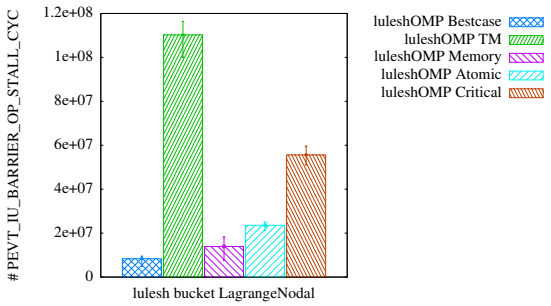


(f) L1P misses with a scrub rate of 6.

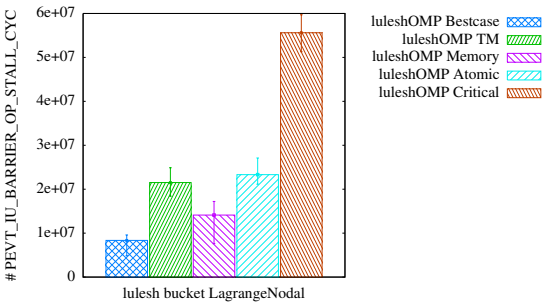
Figure 9.18: BGPM events of LULESH with different scrub rates and long-running TM mode.

In a setup with 64 threads each core executes 4 threads due to the hyper-threading technology. Hence, resources are shared among threads so that 4 threads compete for the private L1 cache. In the following experiments, we are going to reduce the number of parallel threads to 16 so that each core executes only one thread. This frees the experiment of effects caused by the sharing of resources through hyper-threading because threads are assigned to cores in a round-robin fashion so that each core only executes one thread. This experiment helps us to obtain a better understanding of the influence of hyper-threading on the utilization of the microarchitecture.

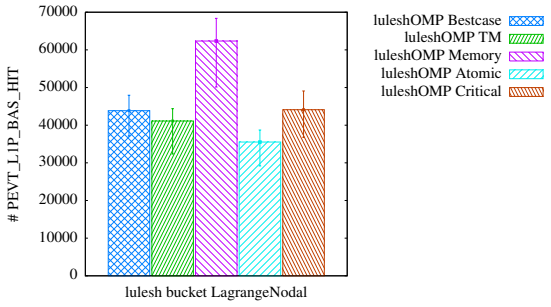
Figure 9.20(a) illustrates that even with only 16 threads and a scrub rate of 66 *TM* experiences more L1P misses than *Bestcase* or *Memory* when running in the short running mode. Figure 9.20(b) confirms this observation with a reduced scrub rate of 6.



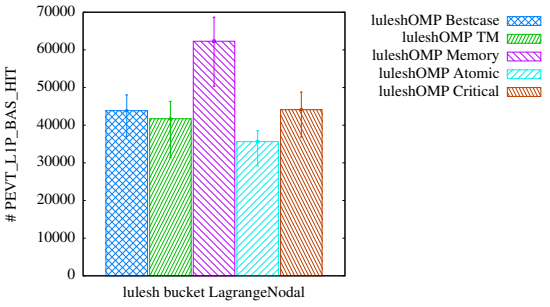
(a) Stall cycles with a scrub rate of 66.



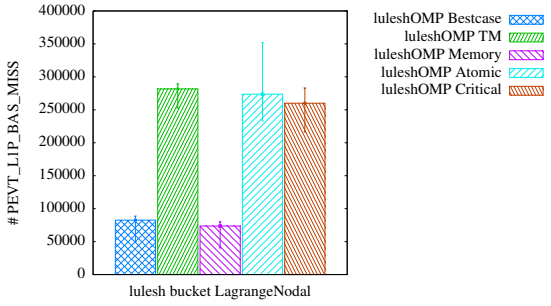
(b) Stall cycles with a scrub rate of 6.



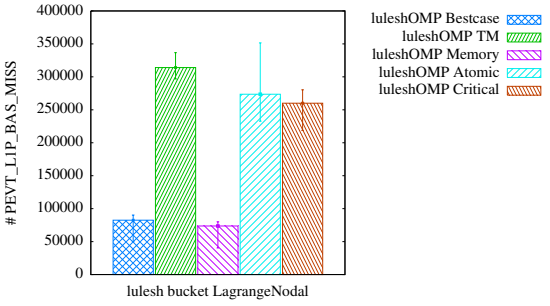
(c) L1P hits with a scrub rate of 66.



(d) L1P hits with a scrub rate of 6.

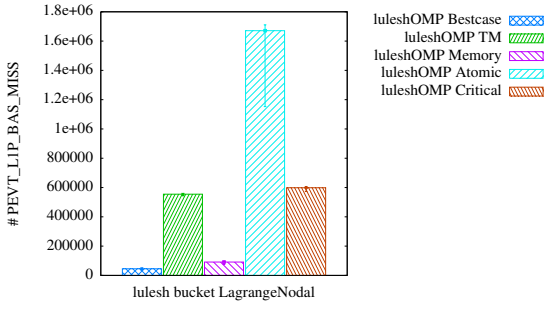


(e) L1P misses with a scrub rate of 66.

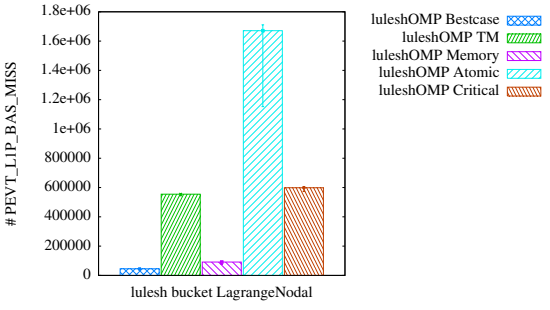


(f) L1P misses with a scrub rate of 6.

Figure 9.19: LULESH with BGPM events executed with different scrub rates and short-running TM mode.

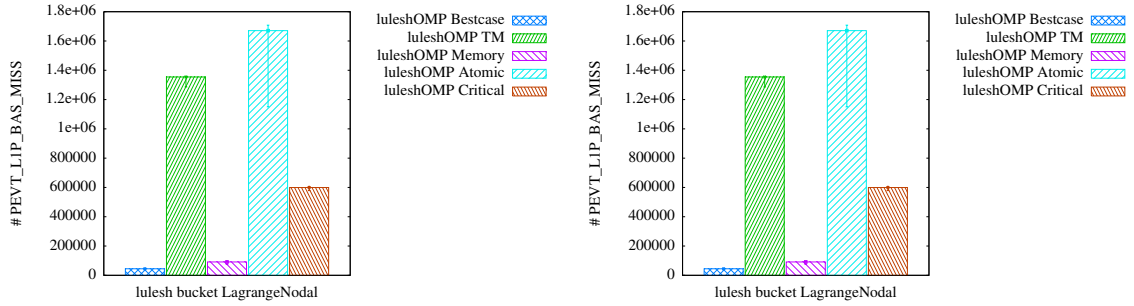


(a) L1P misses with a scrub rate of 66.



(b) L1P misses with a scrub rate of 6.

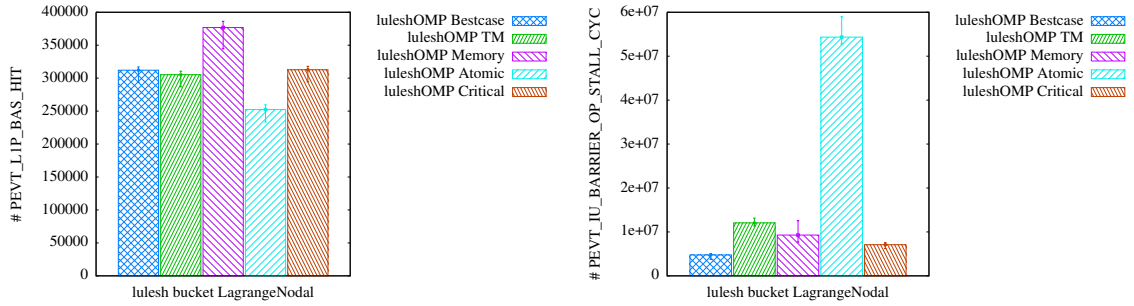
Figure 9.20: L1P misses with LULESH and 16 threads executed in short-running mode.



(a) L1P misses with a scrub rate of 66.

(b) L1P misses with a scrub rate of 6.

Figure 9.21: L1P misses with LULESH and 16 threads executed in long-running mode.



(a) L1P hits with 16 threads.

(b) Stall cycles with 16 threads.

Figure 9.22: L1P hits and stall cycles with LULESH and 16 threads executed in short-running mode with a scrub rate of 6.

Figure 9.21(a) and Figure 9.21(b) show that in the long-running mode *TM* again yields a significantly higher number of L1P misses than *Bestcase* or *Memory*.

A comparison of Figure 9.20 and Figure 9.21 reveals that even with 16 threads the *TM* execution in the short-running mode experiences significantly less L1P misses than *TM* execution in the long-running mode due to the cache flush at the begin of a transaction.

Figure 9.22(a) shows the L1P hits for the short-running mode and a scrub rate of 6 with 16 threads. *TM* has less hits than *Bestcase* or *Memory* with a larger difference. Further, the stall cycles of *TM*, as shown in Figure 9.22(b), are higher than the stall cycles of *Bestcase* with a large difference or *Memory* with a small difference.

These findings illustrate that the reason for the additional L1P misses with TM is not due to the sharing of resources introduced by the hyper-threading technology. Because *Bestcase* and *TM* execute the same memory accesses (but *Bestcase* performs no synchronization), the additional L1P misses are introduced by the TM mechanisms and prohibit a faster run time. Further, both TM modes show a higher L1P miss count but only the long-running mode flushes the L1 cache. Either a data structure in LULESH maps to the same line as a data structure required for TM execution (e.g., transaction handle) or LULESH with *TM* suffers from similar perturbations of the prefetching as we have seen in Section 9.3.6. While the latter is an issue with the TM software stack, the former requires actions of the programmer to rearrange the data structures.

Summarizing the Findings with LULESH

The insights from running LULESH are that the long-running mode does not benefit from setting the scrub rate to a reduced value but the short-running mode yields a speedup of 1.96 for the execution with TM. Still, the *Bestcase* and *Memory* versions of LULESH are faster. We found that the reason for this low performance is the same for both TM modes. For both execution modes, the higher rate of L1P misses causes the performance degradation with TM. This is surprising because the long-running mode flushes the L1 cache whereas the short-running mode by-passes the L1 on stores only. Through further investigations we excluded the use of the hyper-threading technology as a source of contention for the L1 cache. From further performance data we deduce that either LULESH with TM triggers the same prefetching issue as we have seen before, or application data structures map to the same locations in L1 cache as TM-specific data structures and cause the contention. In either case, the use of the profiling tool for TM has been extremely helpful in identifying the issue and narrowing down the causes.

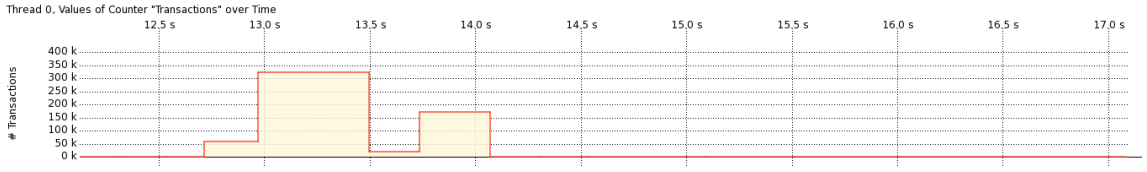
9.5 A Case Study with Vampir Visualizing TM Performance Data

As discussed earlier, the tracing tool can write trace files in the OTF format and that allows visualizations in tools like Vampir. An example visualizing CLOMP-TM performance data is shown in Figure 9.23. CLOMP-TM has been executed with the tracing tool to obtain the event traces. The strength of Vampir is the variety of views that enables a user to compose the performance data in a way that it is most effective for him or her.

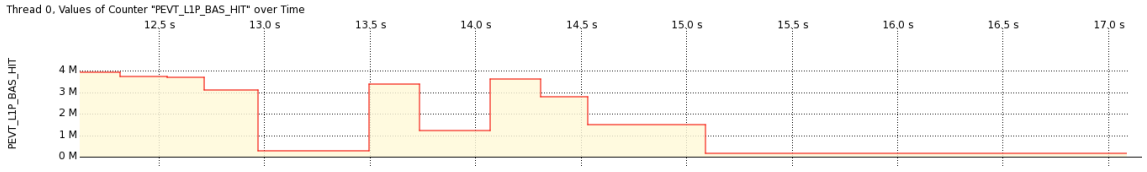
Figure 9.23(a) illustrates the simple but expressive view of one thread that, in this case, shows the amount of transactions executed over time as a rate, in this case executing $400\,000$ transactions per second. This view can be configured to show the differences (as it does now), the accumulated values to the next or the last reading and the values at the measurement points. With these options, the user will find a suitable way of representing the data. In this particular figure the first plateau shows the warm up phase before the *Small TM* execution. CLOMP-TM inserts these warmup phases outside of the timed loops to fill the instruction and data caches to avoid cold start cache effects. The longer plateau up to second 13.5 is the execution of *Small TM*. Then, the warm up phase of *Large TM* follows. The execution of *Large TM* completes at second 14.1. After that CLOMP-TM executes the *Small Critical* and *Large Critical* variants that do not execute transactions.

Figure 9.23(b) shows the rate of the hits in L1P for thread 0. Comparing Figure 9.23(a) and Figure 9.23(b) reveals that the execution of the transactions in *Small TM* and *Large TM* is directly correlated with a low hit rate in the L1P. Figure 9.23(c) confirms this observation for the first 5 threads. The different way of representing the information is called performance radar in Vampir. The color of a field indicates the rate of the event for the corresponding thread. With this simple scheme the information density is higher than with a simple plot over time. In this particular case the plot reveals another interesting aspect: the better L1P hit rate during the warmup phases of *Small TM* and *Large TM* are unique to thread 0. This means that these hits stem from the sequential work (e.g., printing to the screen) that is done by thread 0 only.

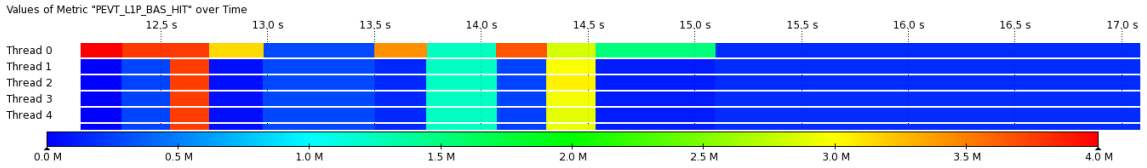
Figure 9.23(d) complements the picture by also presenting higher L1P miss rates during the execution of *Small TM* and *Large TM*. The reason for these misses and the lack of



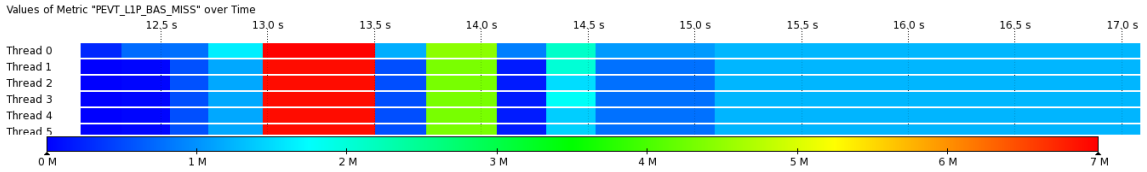
(a) Transactions.



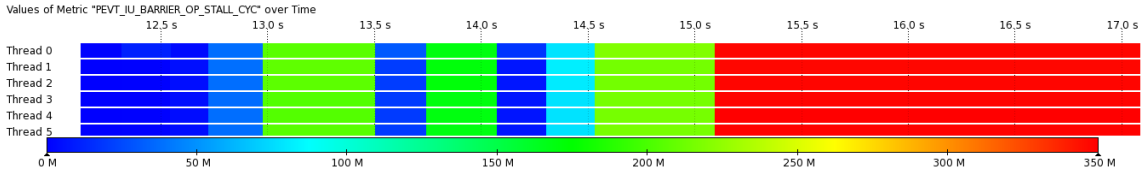
(b) L1P Hits of TM and critical sections for thread 0.



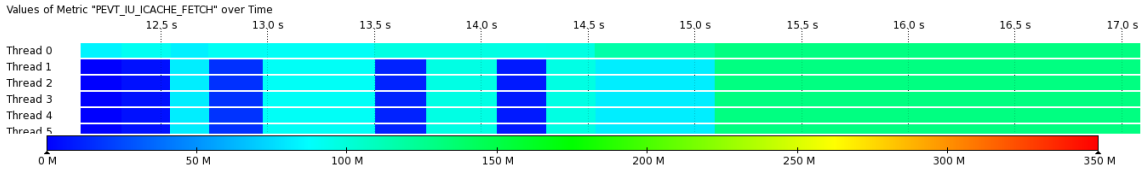
(c) L1P Hits for the first 5 threads.



(d) L1P Misses.



(e) Barrier stall cycles.



(f) Instruction cache fetches with transactions and critical sections.

Figure 9.23: Vampir visualizing the application's behavior with counter plots over time and the performance radar. Run with `clomp_tm_divide1 -1 1 256 128 256 stride1,1,stride1%/2 3 1 0 6 100` in the long-running mode.

hits is the L1 cache flush at the begin of a transaction when the TM system works in the long-running mode. According to the presented performance data, the L1 cache flush includes flushing the L1P. Thus, subsequent L1P requests miss. Figure 9.23(d) illustrates that *Small TM* yields a higher miss rate (and also a lower hit rate) than *Large TM*. The prefetching in *Large TM* has more time to prefetch data because the transaction updates more memory locations and takes longer. Thus, the penalty is not as high as with *Small TM* where, after updating one memory location, the next transaction will again flush the L1 cache.

Figure 9.23(e) illustrates that *Small TM* as well as *Large TM* suffer from a similar rate of stall cycles but yield less stall cycles than the synchronization variants using `omp critical`. Additionally, Figure 9.23(f) shows the rate of instruction fetches and identifies the demanding parts of the benchmark. Moreover, due to the spinning of the threads, the synchronization with `omp critical` has a higher fetch rate than the TM parts (where *Large TM* also shows a slightly higher rate than *Small TM*).

9.6 State of the Art

A profiling method based on transactional events for BG/Q has recently been proposed by Gaudet et al. in [71]. In this one page abstract they base their profiling on the logging of time-stamped, transactional events. The goal is to develop a heuristic that adjusts the serialization threshold to the execution of the application. A handicap of this approach is that events may be lost on a rollback of a transaction. Further, they keep events in a thread-local buffer and write them to a file when the program finishes. As a result they state that the rate of transactional events is very important to compare the two execution modes. The long running mode has a higher rate of events that lead to a faster execution. Compared with our approach, their approach tracks events at a fine granularity and may run into memory problems when an application executes millions of transactions per thread. Our profiling approach aggregates data at user-defined boundaries and mostly has constant memory requirements. Further, their approach cannot help to identify the cause of a slow rate of events. Our approach pairs the transactional statistics with the readings of performance counters so that we can give a qualitative answer as to why specific events are delayed and propose appropriate counter measures.

So far, the remaining tools for TM presented in Chapter 3.4 are not yet ready for the HTM on the BG/Q architecture. The reason is that the TM run time system is proprietary and most tools require some sort of instrumentation of the run time system to gather the performance data. Other techniques, e.g., sampling, cause problems with the interaction of the TM run time so that a sampling action in a transaction may either freeze or change the execution mode of the transaction through a jail mode violation. Then the user observes a behavior of the transaction that would be altered through the sampling and, hence, different from the original execution. Moreover, these tools do not yet support hybrid parallelization that uses message passing in addition to threads.

Up to now, traditional analysis tools, such as the ones described in the following, do not support TM. Score-P is a measurement infrastructure that supports profiling, event trace recording, and online analysis of HPC applications at petascale [114]. Other tools, that also compose with Score-P, are Periscope [74], Scalasca [73], Vampir [143], and TAU [183]. Other popular tools for HPC that use sampling are Open|SpeedShop [179] and HPCTOOLKIT[2]. Thus, our tools close the gap between HPC tools without TM support on one side and highly specialized TM tools without the fitness for BG/Q on the other side.

9.7 Conclusion

In this chapter, we introduce three TM-centric tools that enable the user to gain in-depth insights into the parallel execution at the thread-level. A profiling tool, a tracing tool and a tool designed for determining the overheads of TM execution provide the application developer with complementary information. The three tools access BG/Q performance counters directly through the BGPM interface and complement these readings with statistics from the TM run time. We demonstrate that the overhead tool is capable breaking down the cycles associated with the different execution phases of a transaction. With the profiling tool, we understand the side effects of performing computation before or inside of transactions with respect to the TM execution mode and the utilization of the prefetching unit. We investigate the influence of changing the scrub rate on a variety of BGPM events and explain how to interpret the readings. After identifying key BGPM performance counter events that enable to identify performance issues with TM quickly, we use the profiling tool to uncover the performance issue of TM in the LULESH hydrodynamics proxy application. Further, the tracing tool enables us to visualize the CLOMP-TM benchmark in a time line view and highlight the subtle interaction of the threads. Overall, we see that tool support for TM on BG/Q helps identifying the reasons for degraded performance and supports tuning the right parameters (e.g., scrub rate).

10. Conclusion and Future Work

This Chapter 10 concludes the thesis in Section 10.1 by summarizing and concluding each topic. Section 10.2 present some ideas about future work.

10.1 Summary and Conclusion

In the following, we shortly summarize the findings from optimizing TM applications that either use STM, hybrid TM or HTM.

10.1.1 Information Retrieval for Hybrid TM and STM

In order to retrieve information about the run time behavior of TM applications with hybrid TM and STM, we presented two TM-specific solutions. These capture and preserve the TM application's behavior for STM on an `x86` architecture and hybrid TM on an FPGA through generating event traces.

For the STM solution, TM events are logged, buffered and compressed inside a word-based Software Transactional Memory library. This approach substantially increases the throughput and reduces the application disturbance in comparison with a state-of-the-art binary translation tool (Pin). The more sophisticated trace generation variants employ compression algorithms to reduce the amount of data to be written. The ZLIB and the LZO compression schemes are compared with non-compressing variants. The results show that especially adding dedicated compression threads does have benefits: for large data sets the influence on the run time is reduced significantly. The trace data is compressed with a factor of up to 10. The compression ratio of the ZLIB algorithm is superior to LZO, but also leads to an increased run time for large data sets. Further, LZO has a small influence on the application behavior compared with ZLIB due to its multi-threaded trace compression implementation. For capturing the genuine TM application's behavior this low-intrusiveness is the most important advantage of the developed LZO scheme. Thus, retrieving information throughout this thesis is either done with LZO compression and minimum of 2 compression threads or without compression and a buffer size of 100 K or 1 M elements.

The presented FPGA-based tracing approach augments the hybrid TM system (TMbox). An Event Generation and a Log Unit extend each processor core in order to track changes of the TM state. The event is then transferred during the idle times on a secondary ring bus in order to minimize the intrusiveness. To also monitor software execution of the hybrid TM system, an additional instruction triggers the generation of a log event. This event takes the usual route through the hardware. This approach achieves a continuous stream of events that resembles the application's behavior. The additional hardware requirements are modest and the run time overhead is limited to one additional instruction to monitor the software execution (per event).

A comparison between tracing the software execution of transactions on the FPGA platform and the x86 host deepens the understanding of the techniques. A general and expected trend is that the overall influence on the run time is lower for the hardware-assisted tracing. As a result some benchmarks show a low influence with *STM FPGA*-based tracing and a high influence with *STM x86*-based tracing (e.g., `ssca2` and `intruder`). `genome` with 2 threads also shows a higher influence with *STM FPGA* compared to *STM x86*. In general using the *STM FPGA* machinery for generating traces is preferable because of its lower run time overhead and lower intrusiveness. In cases where an architecture with an FPGA is not available, the *STM x86* approach has also been shown to have a low influence on the run time and comes with the advantage of being portable without requiring programmable hardware.

10.1.2 Optimization of TM Applications

We presented a framework for the Visualization and Optimization of TM Applications (VisOTMA) that provides tools for STM and hybrid TM. VisOTMA supports the experienced as well as the untrained programmer of TM applications when designing, rating and optimizing a TM application. We discussed the question of a profitable transaction length for STM in detail. In order to guide the programmer, we presented a reference application and an optimization algorithm. Further, we employ techniques to capture TM events and dynamic memory requests. The resulting log files enable a comprehensive post-processing and visualization process even for unmanaged languages e.g., `C` or `C++`. Thus, our approach improves the existing solutions through providing a correlation of a TM event with the line in the source code and a mapping of addresses in transactional loads and stores to data structures of the application and combining transactional events with the readings of hardware performance counters. Especially, combining the visualization (Paraver) with the comprehensive transactional statistics inside the VisOTMA framework is useful to uncover bottlenecks of TM applications. We demonstrate the ability of the VisOTMA framework to identify the sources of conflicts in two well-known pathological TM cases (StarvingElder and FriendlyFire). We show how an inexperienced programmer may optimize the TM application even in the absence of performance-critical patterns. This is achieved by tuning the transaction size according to a metric provided by the VisOTMA framework. Through simply enlarging the transactions in the absence of contention, an inexperienced programmer can tune a `C++` application simulating a fluid flow and yield a speedup up to 1.43 over the intuitive transactional version.

STM and the Method of Conjugate Gradients

We implemented the numerical method of Conjugate Gradients without preconditioning in two variants: a normal and a pipelined version. Both use OpenMP critical, OpenMP

atomic, the OpenMP standard reduction and STM to implement the reductions in the algorithm. Our first finding is that the right way of organizing the reductions is the key to performance. A reduction implemented with direct updates of the shared variable will not yield a speedup over execution with one thread regardless of the synchronization primitive. Instead thread-local variables that hold intermediate results are a requirement to achieve speedups. Moreover, the pipelined CG with larger transactions (three times the size) is a strong competitor for normal CG because the number of aborts is modest up to 16 threads. As a downside, pipelined CG required one more iteration to achieve convergence compared with normal CG for our example case. For both CG variants, the wait time at the barriers dominates the time for synchronization in the reduction operations. This also undermines the gains of the execution as well as those due to the optimization of TM. The regular problem structure of CG demands that barriers synchronize all threads after a step in the loop. Thus, a thread that executes a transaction and forces another thread to abort and execute again, simply waits longer at the next barrier for the remaining threads. This basic scenario still holds for longer transactions with pipelined CG. As a result, the CG algorithm is not suited to demonstrate a performance gain with STM. On the other hand, the competitive execution time of pipelined CG with larger transactions and still moderate contention confirms the basic idea of optimizing the TM behavior through employing larger transactions. Moreover, the large difference in execution time for transactions and barriers suggests that future research should target more efficient barrier synchronization or techniques to elide barriers. Common to both CG variants, we found through the use of the hardware performance counters that higher thread counts lead to more L2 cache misses that hinder the scalability and that loads and stores contribute the largest amount to all kinds of instructions retired.

Phase Detection in TM Applications

Embedded in the VisOTMA framework, we presented a systematic approach to detect phase behavior in transactional memory applications. The Transactional Memory Phase Detector (TMPD) is introduced together with two phase detection algorithms adjusted for TM: *Signal Analysis* and *Wavelet Transform*. The results show that we succeed to identify phase behavior in an artificial show case as well as in the STAMP transactional memory benchmarks with both algorithms. We studied the influence of the parameters on the detected phase changes and found that small threshold values and short windows yield the highest detection rate. This is not surprising but also shows that the phases are not very long and differences between the phases are low. These findings indicate that exploiting these phase changes to achieve performance gains, e.g., through changing the parameter setting of the STM, will be a challenging task for STMs.

EigenOpt

To overcome the deficiencies of the current optimization cycle with TM applications that requires an inexperienced programmer to follow a trial-and-error process, we presented EigenOpt. EigenOpt uses the parameterizable EigenBench microbenchmark to simulate the change in application behavior through changes in the parameter set. We extend the tracing machinery and focus the use of performance counters to retrieve the required EigenBench parameters of an arbitrary TM application. These parameters are then used for simulating the outcome of potential optimizations with EigenOpt. This approach has the potential to identify and avoid optimizations with diminishing returns so that it alleviates the optimization process for the inexperienced programmer. However, our findings also

show that the intrusiveness of the tracing approach is high due to the frequent readings of the hardware performance counters and negatively influences the quality of the obtained parameters. This prohibits the application of EigenOpt to a TM application. Future work should investigate the sampling of the hardware performance counters to reduce the intrusiveness and increase the quality of the parameters.

10.1.3 Hybrid TM

We integrated a hybrid TM system, the extended TMbox architecture, in the VisOTMA framework. Visualizing the event traces from executing a network intrusion detection benchmark on TMbox uncovers that this benchmark exhibits multiple pathological execution patterns e.g., Starving Elder, killer transaction and repeated aborts, that are embedded in two application phases. A subsequent optimization of the STM-only execution of the intruder benchmark reveals that a hybrid version with 16 entries in the TM cache can be slower than STM. With 64 entries in the TM cache the execution time is faster than with STM-only. Thread 2 suffers from long, aborted transactions running in software that goes along with a high amount of wasted work. The commit time locking strategy of the STM detects conflicts late so that switching to encounter time locking reduces the amount of wasted work. The execution time decreases from the STM-only version to the hybrid version with encounter time locking by $24.1\,\%$. This performance gain demonstrates that the visualization and the statistics of the run time behavior of a hybrid TM application are suited to tune the execution of a hybrid TM system.

10.1.4 Compilation and Static Information

TM support for GCC

We presented an initial transactional memory extension of the GNU Compiler Collection, and stressed its language-independent and STM-oriented design (yet compatible with hybrid hardware/software implementations). This integration also provides the foundation to integrate TM in an enhanced automatic parallelization strategy, where much of its design and implementation can be reused for the parallelization of sparse, generalized reductions. For these we demonstrate possible performance gains with a manual version. We also highlighted key optimization challenges and opportunities; together with Yoo et al. [214] and the more pessimistic study of Cascaval et al. [27], we stress the importance of compiler and joint language-compiler studies for the future adoption of TM in real world applications. To assess the quality of the ongoing work on the transactional memory branch (that originates from the presented GTM design) by Red Hat Inc., we study the evolution of the compiler optimizations and their impact on the performance of the TM application. We conclude that GCC's compiler and the libITM run time system already made very good progress in tackling the overheads associated with STM.

Conclusion and Outlook for MAPT

In order to exploit the static information available in the compiler to select suited STM parameters, we presented a novel approach for detection and analysis of memory access patterns in transactions. The MAPT approach enhances the LLVM compiler framework and proposes an STM conflict detection granularity to the programmer. These mechanisms are evaluate with test cases and benchmarks from the STAMP benchmark suite. The application run time yields a speedup, yielding a relative improvement in execution time of

$14.7\,\%$ for a transactional K-means clustering algorithm and $16.9\,\%$ for learning a Bayesian network implemented with transactions. The results are promising and validate that an improved throughput can be achieved for the test cases as well as a reduced execution time is possible for the two benchmarks.

10.1.5 HTM of BG/Q from an Application's Perspective

As a HTM system, we evaluated BG/Q's TM hardware from the perspective of an application developer. We introduced CLOMP-TM, a benchmark designed to represent scientific applications, and used it to contrast HTM against traditional synchronization primitives, such as *omp atomic* and *omp critical*. We then extended CLOMP-TM with MPI to mimic hybrid MPI/OpenMP parallelization. Additionally, we studied the impact of environment variables on the performance. Finally, we condensed the findings into a set of best practices and applied them to a Monte Carlo Benchmark. An optimized TM version of MCB with 64 threads achieved a speedup of 27.45 over the baseline. Further, an optimized TM version of the Smoothed Particle Hydrodynamics method from the PARSEC suite achieved a speedup of 14.5 with 64 threads and significantly outperformed a simple TM version (speedup of 4.4) as well as a coarse grain lock (speedup below 1) and verified the usefulness of the best practices. Moreover, our results also show that an expert-level use of lightweight efficient fine-grained locks is hard to beat with TM. For MCB, a synchronization pattern that combines omp atomic and omp critical achieved a slightly larger speedup of 27.57 over baseline. These findings illustrate that performance with HTM does not come for free and, even when following the guidelines developed and presented in Chapter 8, performance with TM may not quite measure up to the expert-level use of locks for the scientific applications considered. However, TM comes with the advantage of improving the programmability and productivity because the user does not have to explicitly manage locks (which is known to be error-prone). Thus, the use of TM or locks depends on the expected gain when comparing development effort with performance improvements.

10.1.6 Tool Support for TM on BG/Q

Further, we introduced three TM-centric tools that enable the user to gain in-depth insights into the parallel execution at the thread-level on BG/Q. A profiling tool, a tracing tool and a tool designed for determining the overheads of TM execution provide the application developer with complementary information. The three tools access BG/Q performance counters directly through the BGPM interface and complement these readings with statistics from the TM run time. We demonstrate that the overhead tool is capable of breaking down the cycles associated with the different execution phases of a transaction. With the profiling tool, we understand the side effects of performing computation before or inside of transactions with respect to the TM execution mode and the utilization of the prefetching unit. We investigate the influence of changing the scrub rate on a variety of BGPM events and explain how to interpret the readings. After identifying key BGPM performance counter events that enable to identify performance issues with TM quickly, we use the profiling tool to uncover the performance issue of TM in the LULESH hydrodynamics proxy application. Further, the tracing tool enables us to visualize the CLOMP-TM benchmark in a time line view and highlight the subtle interaction of the threads. Overall, we see that tool support for TM on BG/Q helps identifying the reasons for degraded performance and supports tuning the right parameters (e.g., scrub rate).

10.2 Outlook and Future Work

Comparing the techniques for retrieving information from the TM run time system, the methods applicable to hybrid TM/STM and HTM differ. While hybrid TM and STM support the tracing of transactional events, the tools designed for the HTM system of BG/Q either trace snapshots or profile the TM application. The coarser granularity of information with HTM does not allow to e.g., track conflicting memory accesses. Since tracking conflicts simplifies to identify a heavily contented data structure, ways to identify conflicts with a proprietary HTM system should be researched in the future. Together with the conflicts, the corresponding data structures should be identified in order to map the conflict back into the space of the application. Then the application developer may rearrange the data structure to minimize contention.

Future work should also concentrate on the research of algorithms that avoid synchronization e.g., through using additional memory or performing extra computations. The first findings with these algorithms, e.g., `luleshOMP Memory`, show promising results and may change the way programs are tuned for performance. The performance tuning of hybrid TM and HTM has similarities: While hybrid TM requires to set parameters such as the number of entries in the TM cache to a suited value, HTM requires to set the right environment variables e.g., scrub rate. In both cases the TM application defines whether this optimization yields performance gains but the application is not changed itself. Future studies should also focus on understanding the implications of structural changes of the TM application and parameter settings on the performance.

Moreover, the phase detection techniques that have been applied to STM so far should be transferred to hybrid TM. Hybrid TM qualifies for phase detection because it has more parameters to adjust (e.g., entries in the TM cache that could be reconfigured at run time) and a measurable performance difference. Phase detection algorithms could be implemented in hardware so that they run in parallel with the application and the high performance gains seem better suited for exploiting this phase behavior.

For the TM-enabled GCC, we identified two directions for the future. First, much of the GTM design and implementation should be reused for the enhanced automatic parallelization strategy, that targets the parallelization of sparse generalized reductions. For these, we demonstrated possible performance gains with a manual version. Moreover, the TM-specific compiler optimizations should be complemented with TM-specific link-time optimizations that may infer whether a symbol has been exported and with this knowledge enable or disable the eliding of TM barriers connected with the symbol.

As more microprocessors with HTM support become available, the techniques presented, in this thesis, should be refined and extended towards these architectures. A potential target is Intel®'s next processor generation, called Haswell, that features Transactional Synchronization Extensions. Due to differences in the implementation, comparing the performance of the HTM subsystem of IBM's Blue Gene/Q and Intel®'s TSX could yield new insights into the design of hardware support for TM and its integration into existing microprocessors in the future.

Bibliography

[1] Adams, M.D., Kossentini, F.: JasPer: a Software-based JPEG-2000 Codec Implementation. In: *International Conference on Image Processing*, Volume 2, September 2000, Pages 53–56, ISSN 1522-4880.

[2] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, Volume 22, Number 6, April 2010, John Wiley and Sons Ltd., Pages 685–701 `http://hpctoolkit.org`, ISSN 1532-0626.

[3] Adl-Tabatabai, A.R., Kozyrakis, C., Saha, B.: Unlocking Concurrency. *Queue*, Volume 4, Number 10, 2007, ACM Press, Pages 24–33, ISSN 1542-7730.

[4] Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and Runtime Support for Efficient Software Transactional Memory. In: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06, New York, NY, USA, ACM, 2006, Pages 26–37, ISBN 1-59593-320-4.

[5] Adl-Tabatabai, A.R., Shpeisman, T., Gottschlich, J.: Draft Specification of Transactional Language Constructs for C++. Online last accessed February 2013, published February 2012 , Version 1.1, Transactional Memory Specification Drafting Group, `https://sites.google.com/site/tmforcplusplus/C%2B%2BTransactionalConstructs-1.1.pdf?attredirects=0`.

[6] Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded Transactional Memory. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. HPCA '05, Washington, DC, USA, IEEE Computer Society, 2005, Pages 316–327, ISBN 0-7695-2275-0.

[7] Ansari, M., Jarvis, K., Kotselidis, C., Luján, M., Kirkham, C., Watson, I.: Profiling Transactional Memory Applications. In: *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. PDP '09, Washington, DC, USA, IEEE Computer Society, 2009, Pages 11–20, ISBN 978-0-7695-3544-9.

[8] Bacon, D., Bloch, J., Bogda, J., Click, C., Haahr, P., Lea, D., May, T., Maessen, J.W., Manson, J., Mitchell, J.D., Nilsen, K., Pugh, B., Sirer, E.G.: The "Double-Checked Locking is Broken" Declaration – Online last accessed July 2012 `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

[9] Baek, W., Minh, C.C., Trautmann, M., Kozyrakis, C., Olukotun, K.: The OpenTM Transactional Application Programming Interface. In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07, Washington, DC, USA, IEEE Computer Society, 2007, Pages 376–387, ISBN 0-7695-2944-5.

[10] Bai, T., Shen, X., Zhang, C., Scherer, W., Ding, C., Scott, M.: A Key-based Adaptive Transactional Memory Executor. In: *IEEE International Parallel and Distributed Processing Symposium*. IPDPS '07, IEEE March 2007, Pages 1–8, ISBN 1-4244-0910-1.

[11] Balasubramonian, R., Albonesi, D., Buyuktosunoglu, A., Dwarkadas, S.: Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In: *MICRO 33: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, New York, NY, USA, ACM, 2000, Pages 245–257, ISBN 1-58113-196-8.

[12] Balasubramonian, R., Dwarkadas, S., Albonesi, D.H.: Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA '03, New York, NY, USA, ACM, 2003, Pages 275–287, ISBN 0-7695-1945-8.

[13] Ball, T., Larus, J.R.: Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, Volume 16, Issue 4, July 1994, ACM, Pages 1319–1360, ISSN 0164-0925.

[14] Barna L. Bihari: Applicability of Transactional Memory to Modern Codes. In: *International Conference on Numerical Analysis and Applied Mathematics*. ICNAAM 10, Rodos, Greece, APS, 2010, Pages 1764–1767, ISBN 978-0-7354-0834-0.

[15] Bihari, B., Wong, M., Wang, A., de Supinski, B., Chen, W.: A Case for Including Transactions in OpenMP II: Hardware Transactional Memory. In Chapman, B., Massaioli, F., Müller, M., Rorro, M., eds.: *OpenMP in a Heterogeneous World*, Volume 7312 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg 2012, Pages 44–58, ISBN 978-3-642-30960-1.

[16] Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing Transactions: The Subtleties of Atomicity. In: *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005 – Online last accessed February 2013 `http://www.cis.upenn.edu/acg/papers/wddd05_atomic_semantics.pdf`.

[17] Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M., Wood, D.A.: Performance Pathologies in Hardware Transactional Memory. *ACM SIGARCH Computer Architecture News*, Volume 35, Issue 2, June 2007, ACM, Pages 81–91, ISSN 0163-5964.

[18] Boehm, H., Gottschlich, J., Luchangco, V., Michael, M., Nelson, C., Riegel, T., Shpeisman, T., Wong, M.: Transactional Language Constructs for C++. Technical Report N3341=12-0031 January 2012.

[19] Börschig, M.: Performance Counter als Hilfsmittel für umfassende Optimierungsstrategien für TM-Anwendungen. Studienarbeit (Study Thesis), Karlsruhe Institute of Technology, Kaiserstr. 12, 76131 Karlsruhe, December 2012.

[20] Bradel, B.J., Abdelrahman, T.S.: The Use of Hardware Transactional Memory for the Trace-Based Parallelization of Recursive Java Programs. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. PPPJ '09, New York, NY, USA, ACM, 2009, Pages 101–110, ISBN 978-1-60558-598-7.

[21] Bronevetsky, G., Gyllenhaal, J., De Supinski, B.R.: CLOMP: Accurately Characterizing OpenMP Application Overheads. In: *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*. IWOMP '08, Berlin, Heidelberg, Springer-Verlag, 2008, Pages 13–25, ISBN 3-540-79560-X, 978-3-540-79560-5.

[22] Bruening, D., Garnett, T., Amarasinghe, S.: An Infrastructure for Adaptive Dynamic Optimization. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '03, Washington, DC, USA, IEEE Computer Society, 2003, Pages 265–275, ISBN 0-7695-1913-X.

[23] Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic Instrumentation of Production Systems. In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '04, Berkeley, CA, USA, USENIX Association, 2004, Pages 2–2.

[24] Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: *Proceedings of the IEEE International Symposium on Workload Characterization*. IISWC '08, IEEE September 2008.

[25] Carey, M.J., DeWitt, D.J., Naughton, J.F.: The OO7 Benchmark. In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD '93, New York, NY, USA, ACM, 1993, Pages 12–21, ISBN 0-89791-592-5.

[26] Casas, M., Badia, R.M., Labarta, J.: Automatic Phase Detection of MPI Applications. In: *Parallel Computing: Architectures, Algorithms and Applications*. NIC Series Volume 38, September 2007 ISBN: 978-3-9810843-4-4.

[27] Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software Transactional Memory: Why is it Only a Research Toy? *Queue*, Volume 6, Number 5, 2008, ACM, Pages 46–58, ISSN 1542-7730.

[28] Casper, J., Oguntebi, T., Hong, S., Bronson, N.G., Kozyrakis, C., Olukotun, K.: Hardware Acceleration of Transactional Memory on Commodity Systems. *ACM SIGARCH Computer Architecture News*, Volume 39, Number 1, March 2011, ACM, Pages 27–38, ISSN 0163-5964.

[29] Castro, M., Georgiev, K., Marangozova-Martin, V., Mehaut, J.F., Fernandes, L.G., Santana, M.: Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures. In: *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. PDP '11, Washington, DC, USA, IEEE Computer Society, 2011, Pages 199–206, ISBN 978-0-7695-4328-4.

[30] Caubet, J., Gimenez, J., Labarta, J., Rose, L.D., Vetter, J.S.: A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In: *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*. WOMPAT '01, London, UK, Springer-Verlag, 2001, Pages 53–67, ISBN 3-540-42346-X.

[31] Chafi, H., Minh, C.C., McDonald, A., Carlstrom, B.D., Chung, J., Hammond, L., Kozyrakis, C., Olukotun, K.: TAPE: A Transactional Application Profiling Environment. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05, New York, NY, USA, ACM, 2005, Pages 199–208, ISBN 1-59593-167-8.

[32] Chakrabarti, D.R., Banerjee, P., Boehm, H.J., Joisha, P.G., Schreiber, R.S.: The Runtime Abort Graph and its Application to Software Transactional Memory Optimization. In: *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11, Washington, DC, USA, IEEE Computer Society, 2011, Pages 42–53, ISBN 978-1-61284-356-8.

[33] Chaudhry, S., Cypher, R., Ekman, M., Karlsson, M., Landin, A., Yip, S., Zeffer, H., Tremblay, M.: Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, Volume 29, Number 2, 2009, Pages 6–16.

[34] Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape Analysis for Java. In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '99, New York, NY, USA, ACM, 1999, Pages 1–19, ISBN 1-58113-238-7.

[35] Christian Bienia: *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[36] Christie, D., Chung, J.W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., Rivière, E.: Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10, New York, NY, USA, ACM, 2010, Pages 27–40, ISBN 978-1-60558-577-2.

[37] Christmann, C., Hebisch, E., Weisbecker, A.: Oversubscription of Computational Resources on Multicore Desktop Systems. In Pankratius, V., Philippsen, M., eds.: *MSEPT '12: International Conference on Multicore Software Engineering, Performance, and Tools*, Volume 7303 of *Lecture Notes in Computer Science*, Springer May 31 - June 1 2012, Pages 18–29, ISBN 978-3-642-31201-4.

[38] Chuck Thacker: Hardware Transactional Memory for Beehive. In: *http://research.microsoft.com/en-us/um/people/birrell/beehive/hardware transactional memory for beehive3.pdf*, MSR Silicon Valley 2010.

[39] Chung, E.S., Nurvitadhi, E., Hoe, J.C., Falsafi, B., Mai, K.: A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs. In: *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. FPGA '08, New York, NY, USA, ACM, 2008, Pages 77–86, ISBN 978-1-59593-934-0.

[40] Chung, E.S., Papamichael, M.K., Nurvitadhi, E., Hoe, J.C., Mai, K., Falsafi, B.: ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FP-

GAs. *ACM Transactions on Reconfigurable Technology and Systems*, Volume 2, Number 2, 2009, Pages 1–32.

[41] Chung, J., Chafi, H., Minh, C., McDonald, A., Carlstrom, B., Kozyrakis, C., Olukotun, K.: The Common Case Transactional Behavior of Multithreaded Programs. In: *The 12th International Symposium on High-Performance Computer Architecture*. HPCA '06, IEEE February 2006, Pages 266–277, ISSN 1530-0897.

[42] Click, C.: Azul's Experiences with Hardware Transactional Memory January 2009 In HP Labs - Bay Area Workshop on Transactional Memory.

[43] Concus, P., Golub, G., O'Leary, D.: *A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations*. Reports Stanford University, Computer Science Department, Stanford University, 1976.

[44] Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, Volume 05, Number 1, 1998, IEEE Computer Society, Pages 46–55, ISSN 1070-9924.

[45] Dalessandro, L., Marathe, V.J., Spear, M.F., Scott, M.L.: Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In: *ACM SIGPLAN Workshop on Transactional Computing*. TRANSACT '07, 2007.

[46] Dalessandro, L., Scott, M.L.: Strong Isolation is a Weak Idea. In: *Proceedings of the Fourth ACM SIGPLAN Workshop on Transactional Computing*. TRANSACT, January 2009.

[47] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid Transactional Memory. *SIGARCH Computer Architecture News*, Volume 34, Issue 5, October 2006, ACM, Pages 336–346, ISSN 0163-5964.

[48] Dave, N., Pellauer, M., Emer, J.: Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA. In: *2nd Workshop on Architecture Research using FPGA Platforms*. WARFP '06, 2006.

[49] Davis, J., Hammond, L., Olukotun, K.: A Flexible Architecture for Simulation and Testing (FAST) Multiprocessor Systems. In: *1st Workshop on Architecture Research using FPGA Platforms*. WARFP '05, 2005.

[50] de Oliveira Stein, B., de Kergommeaux, J.C.: Pajé Trace File Format. Technical report, Universidade Federal de Santa Maria, RS, Brazil and Laboratoire Logiciel Systèmes et Réseaux, France March 2003.

[51] Dean, J., Hicks, J.E., Waldspurger, C.A., Weihl, W.E., Chrysos, G.: ProfileMe: Hardware Support for Instruction-Level Profiling on Out-Of-Order Processors. In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 30, Washington, DC, USA, IEEE Computer Society, 1997, Pages 292–302, ISBN 0-8186-7977-8.

[52] Demsky, B., Dash, A.: Using Discrete Event Simulation to Analyze Contention Managers. *International Journal of Parallel Programming*, Volume 39, Number 6, December 2011, Springer Netherlands, Pages 783–808, ISSN 0885-7458.

[53] Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early Experience with a Commercial Hardware Transactional Memory Implementation. *SIGPLAN Not.*, Volume 44, Issue 3, March 2009, ACM, Pages 157–168, ISSN 0362-1340.

[54] Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: *Proceedings of the 20th International Conference on Distributed Computing*. DISC '06, Berlin, Heidelberg, Springer-Verlag, 2006, Pages 194–208, ISBN 3-540-44624-9, 978-3-540-44624-8.

[55] Ding, C., Dwarkadas, S., Huang, M.C., Shen, K., Carter, J.B.: Program Phase Detection and Exploitation. In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. IPDPS '06, Washington, DC, USA, IEEE Computer Society, 2006, Pages 279–279, ISBN 1-4244-0054-6.

[56] Dragojevic, A., Ni, Y., Adl-Tabatabai, A.R.: Optimizing Transactions for Captured Memory. In: *Proceedings of the twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '09, New York, NY, USA, ACM, 2009, Pages 214–222, ISBN 978-1-60558-606-9.

[57] Duesterwald, E., Cascaval, C., Dwarkadas, S.: Characterizing and Predicting Program Behavior and its Variability. In: *International Conference on Parallel Architectures and Compilation Techniques*, Volume 0 of *PACT '03*, Los Alamitos, CA, USA, IEEE Computer Society, 2003, Pages 220–231, ISSN 1089-795X., ISBN 0-7695-2021-9.

[58] Ennals, R.: Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge Technical Report January 2006.

[59] Esselson, A.: Algorithmen und Metriken zur anwendungsorientierten Auswahl von STM-Parametern. Diplomarbeit (Master's Thesis), Karlsruhe Institute of Technology, Kaiserstr. 12, 76131 Karlsruhe, July 2010.

[60] Faure, E., Benabdenbi, M., Pecheux, F.: Distributed Online Software Monitoring of Manycore Architectures. In: *Proceedings of the 2010 IEEE 16th International On-Line Testing Symposium*. IOLTS '10, Washington, DC, USA, IEEE Computer Society, 2010, Pages 56–61, ISBN 978-1-4244-7724-1.

[61] Felber, P., Fetzer, C., Müller, U., Riegel, T., Süßkraut, M., Sturzrehm, H.: Transactifying Applications using an Open Compiler Framework. In: *Workshop on Transactional Computing*. TRANSACT '07, August 2007.

[62] Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '08, New York, NY, USA, ACM, 2008, Pages 237–246, ISBN 978-1-59593-795-7.

[63] Ferri, C., Moreshet, T., Bahar, R.I., Benini, L., Herlihy, M.: A Hardware/Software Framework for supporting Transactional Memory in a MPSoC Environment. *ACM SIGARCH Computer Architecture News*, Volume 35, Number 1, March 2007, ACM, Pages 47–54, ISSN 0163-5964.

[64] Ferri, C., Wood, S., Moreshet, T., Iris Bahar, R., Herlihy, M.: Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *Journal of Parallel and Distributed Computing*, Volume 70, Number 10, October 2010, Academic Press, Inc., Pages 1042–1052, ISSN 0743-7315.

[65] Fidge, C. J.: Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Proceedings of the 11th Australian Computer Science Conference*, Volume 10, Number 1, 1988, Pages 56–66.

[66] Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and Mixins. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98, New York, NY, USA, ACM, 1998, Pages 171–183, ISBN 0-89791-979-3.

[67] Fraser, K., Harris, T.: Concurrent Programming Without Locks. *ACM Transactions on Computer Systems (TOCS)*, Volume 25, Number 5, Issue 2, May 2007, ACM, ISSN 0734-2071.

[68] Froyd, N., Mellor-Crummey, J., Fowler, R.: Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05, New York, NY, USA, ACM, 2005, Pages 81–90, ISBN 1-59593-167-8.

[69] Gajinov, V., Zyulkyarov, F., Unsal, O.S., Cristal, A., Ayguade, E., Harris, T., Valero, M.: QuakeTM: Parallelizing a complex sequential application using transactional memory. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09, New York, NY, USA, ACM, 2009, Pages 126–135, ISBN 978-1-60558-498-0.

[70] Gamblin, T., de Supinski, B.R., Schulz, M., Fowler, R., Reed, D.A.: Scalable Load-Balance Measurement for SPMD Codes. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08, Piscataway, NJ, USA, IEEE Press, 2008, Pages 1–12, ISBN 978-1-4244-2835-9.

[71] Gaudet, M., Amaral, J.N.: Transactional Event Profiling in a Best-Effort Hardware Transactional Memory System. In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12, New York, NY, USA, ACM, 2012, Pages 475–476, ISBN 978-1-4503-1182-3.

[72] Gauss, A.: Algorithmen zur Detektion von Programmphasen in TM-Anwendungen. Diplomarbeit (Master's Thesis), Karlsruhe Institute of Technology, Kaiserstr. 12, 76131 Karlsruhe, March 2010.

[73] Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The SCALASCA performance toolset architecture, Volume 22, Number 6, April 2010, John Wiley and Sons Ltd., Pages 702–719, ISSN 1532-0626.

[74] Gerndt, M., Ott, M.: Automatic Performance Analysis with Periscope. *Concurrency and Computation: Practice and Experience*, Volume 22, Number 6, April 2010, John Wiley and Sons Ltd., Pages 736–748, ISSN 1532-0626.

[75] Goetz, B.: Optimistic Thread Concurrency – Breaking the Scale Barrier January 2006 Whitepaper from Azul Systems Inc., Online `http://www.pointsource.com/staff/docArchive/WP00081-OptimisticConcurrency-LckngTech-Azul-WP.pdf` last accessed 4th February 2013.

[76] Gottschlich, J.E., Herlihy, M.P., Pokam, G.A., Siek, J.G.: Visualizing Transactional Memory. In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12, New York, NY, USA, ACM, 2012, Pages 159–170, ISBN 978-1-4503-1182-3.

[77] Gottschlich, J.E., Vachharajani, M., Siek, J.G.: An Efficient Software Transactional Memory Using Commit-Time Invalidation. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10, New York, NY, USA, ACM, 2010, Pages 101–110, ISBN 978-1-60558-635-9.

[78] Grinberg, S., Weiss, S.: Investigation of Transactional Memory Using FPGAs. In: *Proceedings of the 2nd Workshop on Architecture Research using FPGA Platforms*. WARFP '06, 2006.

[79] Grossman, D., Manson, J., Pugh, W.: What Do High-Level Memory Models Mean for Transactions? In: *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*. MSPC '06, New York, NY, USA, ACM, 2006, Pages 62–69, ISBN 1-59593-578-9.

[80] Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional Memory Coherence and Consistency. *SIGARCH Computer Architure News*, Volume 32, Issue 2, March 2004, ACM, Pages 102–113, ISSN 0163-5964.

[81] Haring, R.: The Blue Gene/Q Compute Chip. In: *Hot Chips 23*, August 2011 Online last accessed 26th of February 2013 `http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.1-manycore/HC23.18.121.BlueGene-IBM_BQC_HC23_20110818.pdf`.

[82] Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G., Boyle, P., Christ, N., Kim, C.: The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, Volume 32, Number 2, March 2012, IEEE Computer Society Press, Pages 48–60, ISSN 0272-1732.

[83] Harris, T., Larus, J., Rajwar, R.: *Transactional Memory*, Volume 5, Morgan & Claypool Publishers, June 2010, 2nd edition, Synthesis Lectures on Computer Architecture, ISBN 978-1608452354.

[84] Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable Memory Transactions. In: *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05, New York, NY, USA, ACM, 2005, Pages 48–60, ISBN 1-59593-080-9.

[85] Heindl, A., Pokam, G.: An Analytic Framework for Performance Modeling of Software Transactional Memory. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Volume 53, Number 8, 2009, Elsevier North-Holland, Inc., Pages 1202–1214, ISSN 1389-1286.

[86] Hennessy, J.L., Patterson, D.A.: *Computer Architecture: A Quantitative Approach*. 5th edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA October 2011, ISBN 9780123838728.

[87] Herlihy, M., Lev, Y.: tm_db: A Generic Debugging Library for Transactional Programs. In: *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, IEEE Computer Society, 2009, Pages 136–145, ISBN 978-0-7695-3771-9.

[88] Herlihy, M., Luchangco, V., Moir, M.: Obstruction-Free Synchronization: Double-Ended Queues as an Example. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems*. ICDCS '03, Washington, DC, USA, IEEE Computer Society, 2003, Pages 522–, ISBN 0-7695-1920-2.

[89] Herlihy, M., Luchangco, V., Moir, M.: A Flexible Framework for Implementing Software Transactional Memory. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06, New York, NY, USA, ACM, 2006, Pages 253–262, ISBN 1-59593-348-4.

[90] Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software Transactional Memory for Dynamic-Sized Data Structures. In: *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*. PODC '03, New York, NY, USA, ACM, 2003, Pages 92–101, ISBN 1-58113-708-7.

[91] Herlihy, M., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93, New York, NY, USA, ACM, 1993, Pages 289–300, ISBN 0-8186-3810-9.

[92] Heuveline, V., Janko, S., Karl, W., Rocker, B., Schindewolf, M.: Software Transactional Memory, OpenMP and Pthread Implementations of the Conjugate Gradients Method – A Preliminary Evaluation. In Daydé, M., Marques, O., Nakajima, K., eds.: *High Performance Computing for Computational Science - VECPAR 2012*, Volume 7851 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg July 2013, Pages 300–313, ISBN 978-3-642-38717-3.

[93] Hill, M.D., Hower, D., Moore, K.E., Swift, M.M., Volos, H., Wood, D.A.: A Case for Deconstructing Hardware Transactional Memory Systems. In Cohen, A., Garzarán, M.J., Lengauer, C., Midkiff, S.P., eds.: *Programming Models for Ubiquitous Parallelism*, Volume 07361 of *Dagstuhl Seminar Proceedings*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany 2007.

[94] Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: EigenBench: A Simple Exploration Tool for Orthogonal TM Characteristics. In: *Proceedings of the IEEE International Symposium on Workload Characterization*. IISWC '10, Washington, DC, USA, IEEE Computer Society, 2010, Pages 1–11, ISBN 978-1-4244-9297-8.

[95] Huang, M., Renau, J., Torrellas, J.: Profile Based Energy Reduction for High-Performance Processors. In: *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[96] Huang, M.C., Renau, J., Torrellas, J.: Positional Adaptation of Processors: Application to Energy Reduction. In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA '03, New York, NY, USA, ACM, 2003, Pages 157–168, ISBN 0-7695-1945-8.

[97] Hudson, R.L., Saha, B., Adl-Tabatabai, A.R., Hertzberg, B.C.: McRT-Malloc: A Scalable Transactional Memory Allocator. In: *Proceedings of the 5th International Symposium on Memory Management*. ISMM '06, New York, NY, USA, ACM, 2006, Pages 74–83, ISBN 1-59593-221-6.

[98] Huffmire, T., Sherwood, T.: Wavelet-Based Phase Classification. In: *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. PACT '06, New York, NY, USA, ACM, 2006, Pages 95–104, ISBN 1-59593-264-X.

[99] Hughes, C., Li, T.: Optimizing Throughput/Power Trade-offs in Hardware Transactional Memory Using DVFS and Intelligent Scheduling. In: *Proceedings of the International Conference on Supercomputing*. ICS '11, New York, NY, USA, ACM, 2011, Pages 141–150, ISBN 978-1-4503-0102-2.

[100] Hughes, C., Poe, J., Qouneh, A., Li, T.: On the (Dis)similarity of Transactional Memory Workloads. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*. IISWC '09, Washington, DC, USA, IEEE Computer Society, October 2009, Pages 108–117, ISBN 978-1-4244-5156-2.

[101] IBM: IBM XL C/C++ for Transactional Memory for AIX. `http://www.alphaworks.ibm.com/tech/xlcstm` May 2008.

[102] Intel® Corporation: *Technologies for Measuring Software Performance: VTune Analyzers*. 2003 White paper.

[103] Intel® Corporation: *Intel® Architecture Instruction Set Extensions Programming Reference*. February 2012 319433-012A.

[104] İpek, E., Martínez, J.F., de Supinski, B.R., McKee, S.A., Schulz, M.: Dynamic Program Phase Detection in Distributed Shared-Memory Multiprocessors. In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. IPDPS '06, Washington, DC, USA, IEEE Computer Society, 2006, Pages 280–280, ISBN 1-4244-0054-6.

[105] Janko, S.: Multi-threaded Implementations of the Conjugate Gradients Method based on Transactional Memory with OpenMP and Pthreads. Studienarbeit (Study Thesis), Karlsruhe Institute of Technology, Kaiserstr. 12, 76131 Karlsruhe, February 2012.

[106] Janko, S., Rocker, B., Schindewolf, M., Heuveline, V., Karl, W.: Software Transactional Memory, OpenMP and Pthread implementations of the Conjugate Gradients Method - a Preliminary Evaluation. Number 01, EMCL Preprint Series 2012 `http://www.emcl.kit.edu/preprints/emcl-preprint-2012-01.pdf`, ISSN 2191–0693.

[107] Jost, G., Jin, H., Labarta, J., Gimenez, J.: Interfacing computer aided Parallelization and performance analysis. In: *Proceedings of the 2003 International Conference on Computational science*. ICCS'03, Berlin, Heidelberg, Springer-Verlag, 2003, Pages 181–190, ISBN 3-540-40197-0.

[108] Kachris, C., Kulkarni, C.: Configurable Transactional Memory. In: *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, IEEE Computer Society, 2007, Pages 65–72, ISBN 0-7695-2940-2.

[109] Kang, S., Bader, D.A.: An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs. *SIGPLAN Not.*, Volume 44, Number 4, February 2009, ACM, Pages 15–24, ISSN 0362-1340.

[110] Karrels, E., Lusk, E.: Performance Analysis of MPI Programs. In Dongarra, J., Tourancheau, B., eds.: *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*. Proceedings in Applied Mathematics Series, Society for Industrial & Applied Publications 1994, Pages 195–200, ISBN 9780898713435.

[111] Kirchhofer, P.: Enhancing an HTM System with HW Monitoring Capabilities. Studienarbeit (Study Thesis), Karlsruhe Institute of Technology, Kaiserstr. 12, 76131 Karlsruhe, December 2011.

[112] Kirchhofer, P., Schindewolf, M., Sonmez, N., Arcas, O., Unsal, O.S., Cristal, A., Karl, W.: Enhancing an HTM System with Monitoring, Visualization and Analysis Capabilities. In: *Euro-TM Workshop on Transactional Memory (WTM 2012)*, April 2012 Abstract available at `http://www.eurotm.org/action-meetings/wtm2012/program/abstracts#Kirchhofer`.

[113] Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.: Introducing the Open Trace Format (OTF). In Alexandrov, V., van Albada, G., Sloot, P., Dongarra, J., eds.: *Computational Science ICCS 2006*, Volume 3992 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg 2006, Pages 526–533.

[114] Knüpfer, A., Rössel, C., an Mey, D., Biersdorf, S., Diethelm, K., Eschweiler, D., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In Brunst, H., Müller, M.S., Nagel, W.E., Resch, M.M., eds.: *Tools for High Performance Computing 2011*, Springer Berlin Heidelberg 2011, Pages 79–91, ISBN 978-3-642-31475-9.

[115] Labrecque, M., Jeffrey, M.C., Steffan, J.G.: Application-Specific Signatures for Transactional Memory in Soft Processors. *ACM Transactions on Reconfigurable Technology and Systems*, Volume 4, Number 3, Number 21, August 2011, ACM, Pages 21:1–21:14, ISSN 1936-7406.

[116] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, Volume 28, Number 9, September 1979, IEEE Computer Society, Pages 690–691, ISSN 0018-9340.

[117] Larus, J., Kozyrakis, C.: Transactional Memory. *Communications of the ACM*, Volume 51, Number 7, July 2008, ACM, Pages 80–88, ISSN 0001-0782.

[118] Lattner, C.: *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaign, 2002.

[119] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04, Washington, DC, USA, IEEE Computer Society, 2004, Pages 75–, ISBN 0-7695-2102-9.

[120] Lawrence Livermore National Laboratory: Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254.

[121] Lehman, P.L., Yao, S.B.: Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, Volume 6, Number 4, December 1981, ACM, Pages 650–670, ISSN 0362-5915.

[122] Lev, Y., Moir, M., Nussbaum, D.: PhTM: Phased Transactional Memory. In: *2nd Workshop on Transactional Computing*. TRANSACT '07, August 2007 – Online last accessed 4th February 2013 `http://research.sun.com/projects/scalable/pubs/TRANSACT2007-PhTM.pdf`.

[123] Lev, Y.: *Debugging and Profiling of Transactional Programs*. PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island May 2010.

[124] Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a Scalable Software Transactional Memory. In: *Workshop on Transactional Computing*. TRANSACT '09, February 2009.

[125] Lim, M.Y., Freeh, V.W., Lowenthal, D.K.: Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06, New York, NY, USA, ACM, 2006, Pages 107–121, ISBN 0-7695-2700-0.

[126] Lomet, D.B.: Process Structuring, Synchronization, and Recovery using Atomic Actions. In: *Proceedings of an ACM Conference on Language Design for Reliable Software*, New York, NY, USA, ACM, 1977, Pages 128–137.

[127] Lourenço, J., Dias, R., Luís, J., Rebelo, M., Pessanha, V.: Understanding the Behavior of Transactional Memory Applications. In: *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. PADTAD '09, New York, NY, USA, ACM, 2009, Pages 3:1–3:9, ISBN 978-1-60558-655-7.

[128] Luchangco, V.: Against Lock-Based Semantics for Transactional Memory. In: *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '08, New York, NY, USA, ACM, 2008, Pages 98–100, ISBN 978-1-59593-973-9.

[129] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices*, Volume 40, Issue 6, June 2005, ACM, Pages 190–200, ISSN 0362-1340.

[130] Magklis, G., Scott, M.L., Semeraro, G., Albonesi, D.H., Dropsho, S.: Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In: *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, New York, NY, USA, ACM, 2003, Pages 14–27, ISBN 0-7695-1945-8.

[131] Marathe, V.J., Scherer, W.N., Scott, M.L.: Adaptive Software Transactional Memory. In: *Proceedings of the 19th international conference on Distributed Computing*. DISC'05, Berlin, Heidelberg, Springer-Verlag, 2005, Pages 354–368, ISBN 3-540-29163-6, 978-3-540-29163-3.

[132] Marathe, V.J., Scherer III, W.N., Scott, M.L.: Design Tradeoffs in Modern Software Transactional Memory Systems. In: *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*. LCR '04, New York, NY, USA, ACM, Oct 2004, Pages 1–7.

[133] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Practical Weak-Atomicity Semantics for Java STM. In: *Proceedings of*

the 20th Annual Symposium on Parallelism in Algorithms and Architectures. SPAA '08, New York, NY, USA, ACM, 2008, Pages 314–325, ISBN 978-1-59593-973-9.

[134] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Single Global Lock Semantics in a Weakly Atomic STM. *ACM SIGPLAN Notices*, Volume 43, Number 5, May 2008, ACM, Pages 15–26, ISSN 0362-1340.

[135] Meurant, G.: Multitasking the Conjugate Gradient Method on the CRAY X-MP/48. *Parallel Computing*, Volume 5, Number 3, 1987, Pages 267–280.

[136] Milovanović, M., Ferrer, R., Gajinov, V., Unsal, O.S., Cristal, A., Ayguadé, E., Valero, M.: Multithreaded Software Transactional Memory and OpenMP. In: *Proceedings of the Workshop on Memory Performance*. MEDEA '07, New York, NY, USA, ACM, 2007, Pages 81–88, ISBN 978-1-9593-807-7.

[137] Moore, K.F., Grossman, D.: High-Level Small-Step Operational Semantics for Transactions. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08, New York, NY, USA, ACM, 2008, Pages 51–62, ISBN 978-1-59593-689-9.

[138] Moore, K., Bobba, J., Moravan, M., Hill, M., Wood, D.: LogTM: Log-based Transactional Memory. In: *The 12th International Symposium on High-Performance Computer Architecture*. HPCA '06, February 2006, Pages 254–265, ISSN 1530-0897.

[139] Moore, S., Terpstra, D., London, K., Mucci, P., Teller, P., Salayandia, L., Bayona, A., Nieto, M.: PAPI Deployment, Evaluation, and Extensions. In: *Proceedings of the 2003 DoD User Group Conference*. DOD_UGC '03, Washington, DC, USA, IEEE Computer Society, 2003, Pages 349–, ISBN 0-7695-1953-9.

[140] Moreshet, T., Bahar, R.I., Herlihy, M.: Energy Reduction in Multiprocessor Systems Using Transactional Memory. In: *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*. ISLPED '05, New York, NY, USA, ACM, 2005, Pages 331–334, ISBN 1-59593-137-6.

[141] Moreshet, T., Bahar, R.I., Herlihy, M.: Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks. In: *Workshop on Memory Performance Issues*, February 2006, Pages 1–7 held in conjunction with the International Symposium on High-Performance Computer Architecture.

[142] Müller, M., Charypar, D., Gross, M.: Particle-based fluid simulation for interactive Applications. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation*. SCA '03, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association, 2003, Pages 154–159, ISBN 1-58113-659-5.

[143] Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In Bischof, C., Bücker, M., Gibbon, P., Joubert, G., Lippert, T., Mohr, B., Peters, F., eds.: *Parallel Computing: Architectures, Algorithms and Applications*, Volume 15 of *Advances in Parallel Computing*, IOS Press 2007, Pages 637–644 ISBN 978-1-58603-796-3.

[144] Nandy, S., Gao, X., Ferrante, J.: TFP: Time-Sensitive, Flow-Specific Profiling at Runtime. In Rauchwerger, Lawrence, ed.: *Languages and Compilers for Parallel Computing*, Volume 2958 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg 2004, Pages 32–47, ISBN 978-3-540-21199-0.

[145] Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07, New York, NY, USA, ACM, 2007, Pages 89–100, ISBN 978-1-59593-633-2.

[146] Nielsen, O.M., Hegland, M.: Parallel Performance of Fast Wavelet Transforms. *International Journal of High Speed Computing*, Volume 11, Number 1, 2000, Pages 55–74, ISSN 0129-0533.

[147] Njoroge, N., Casper, J., Wee, S., Teslyar, Y., Ge, D., Kozyrakis, C., Olukotun, K.: ATLAS: A Chip-Multiprocessor with Transactional Memory Support. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '07, San Jose, CA, USA, EDA Consortium, 2007, Pages 3–8, ISBN 978-3-9810801-2-4.

[148] Oechslein, B.: Statische WCET-Analyse von LLVM-Bytecode. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, August 2008.

[149] Pankratius, V., Adl-Tabatabai, A.R.: A Study of Transactional Memory vs. Locks in Practice. In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11, New York, NY, USA, ACM, 2011, Pages 43–52, ISBN 978-1-4503-0743-7.

[150] Pankratius, V., Adl-Tabatabai, A.R., Otto, F.: Does Transactional Memory Keep Its Promises? Results from an Empirical Study. Technical Report 2009-12, Institute for Programme Structures and Data Organisation, University of Karlsruhe, Germany September 2009.

[151] Pant, S.M., Byrd, G.T.: Limited Early Value Communication to Improve Performance of Transactional Memory. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09, New York, NY, USA, ACM, 2009, Pages 421–429, ISBN 978-1-60558-498-0.

[152] Payer, M., Gross, T.R.: Performance Evaluation of Adaptivity in Software Transactional Memory. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '11, Washington, DC, USA, IEEE Computer Society, 2011, Pages 165–174, ISBN 978-1-61284-367-4.

[153] Perelman, E., Polito, M., Bouguet, J.Y., Sampson, J., Calder, B., Dulong, C.: Detecting Phases in Parallel Applications on Shared Memory Architectures. In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. IPDPS '06, Washington, DC, USA, IEEE Computer Society, 2006, Pages 88–88, ISBN 1-4244-0054-6.

[154] Poe, J., Hughes, C., Li, T.: TransPlant: A Parameterized Methodology For Generating Transactional Memory Workloads. In: *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*. MASCOTS '09, September 2009, Pages 1 –10, ISSN 1526-7539.

[155] Poe, J., Cho, C.B., Li, T.: Using Analytical Models to Efficiently Explore Hardware Transactional Memory and Multi-core Co-design. In: *Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*. SBAC-PAD '08, Washington, DC, USA, IEEE Computer Society, 2008, Pages 159–166, ISBN 978-0-7695-3423-7.

[156] Porter, D.E., Witchel, E.: Understanding Transactional Memory Performance. In: *International Symposium on Performance Analysis of Systems and Software*. ISPASS '10, IEEE March 2010, Pages 97–108, ISBN 978-1-4244-6023-6.

[157] Pusceddu, M., Ceccolini, S., Palermo, G., Sciuto, D., Tumeo, A.: A Compact Transactional Memory Multiprocessor System on FPGA. In: *International Conference on Field Programmable Logic and Applications*. FPL '10, 31 2010-sept. 2 2010, Pages 578–581, ISSN 1946-1488.

[158] Rajwar, R., Herlihy, M., Lai, K.: Virtualizing Transactional Memory. In: *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ISCA '05, Washington, DC, USA, IEEE Computer Society, 2005, Pages 494–505, ISBN 0-7695-2270-X.

[159] Ramadan, H.E., Rossbach, C.J., Witchel, E.: Dependence-Aware Transactional Memory for Increased Concurrency. In: *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41, Washington, DC, USA, IEEE Computer Society, 2008, Pages 246–257, ISBN 978-1-4244-2836-6.

[160] Ramadan, H.E., Roy, I., Herlihy, M., Witchel, E.: Committing Conflicting Transactions in an STM. In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '09, New York, NY, USA, ACM, 2009, Pages 163–172, ISBN 978-1-60558-397-6.

[161] Riegel, T., Becker de Brum, D.: Making Object-Based STM Practical in Unmanaged Environments. In: *Workshop on Transactional Computing*. TRANSACT '08, 2008.

[162] Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: *Proceedings of the 20th International Conference on Distributed Computing*. DISC '06, Berlin, Heidelberg, Springer-Verlag, 2006, Pages 284–298, ISBN 3-540-44624-9, 978-3-540-44624-8.

[163] Rossbach, C.J., Hofmann, O.S., Porter, D.E., Ramadan, H.E., Aditya, B., Witchel, E.: TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. *SIGOPS Operating Systems Review*, Volume 41, Number 6, October 2007, ACM, Pages 87–102, ISSN 0163-5980.

[164] Rossbach, C.J., Hofmann, O.S., Witchel, E.: Is Transactional Programming Actually Easier? *ACM SIGPLAN Notices*, Volume 45, Issue 5, January 2010, ACM, Pages 47–56, ISSN 0362-1340.

[165] Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, Volume 14, Number 2, April 2009, Kluwer Academic Publishers, Pages 131–164, ISSN 1382-3256.

[166] Ruth A. Aydt: The Pablo Self-Defining Data Format. Technical report, Urbana, Illinois 61801, USA 1994.

[167] Saad, Y.: *Iterative Methods for Sparse Linear Systems*. 2nd edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA 2003, ISBN 0898715342.

[168] Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In: *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '06, New York, NY, USA, ACM, 2006, Pages 187–197, ISBN 1-59593-189-9.

[169] Saha, B., Adl-Tabatabai, A.R., Jacobson, Q.: Architectural Support for Software Transactional Memory. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '06, Washington, DC, USA, IEEE Computer Society, 2006, Pages 185–196, ISBN 0-7695-2732-9.

[170] Scherer, III, W.N., Scott, M.L.: Advanced Contention Management for Dynamic Software Transactional Memory. In: *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*. PODC '05, New York, NY, USA, ACM, 2005, Pages 240–248, ISBN 1-58113-994-2.

[171] Schindewolf, M., Bihari, B., Gyllenhaal, J., Schulz, M., Wang, A., Karl, W.: What Scientific Applications Can Benefit from Hardware Transactional Memory? In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press, 2012, Pages 90:1–90:11, ISBN 978-1-4673-0804-5.

[172] Schindewolf, M., Cohen, A., Karl, W., Marongiu, A., Benini, L.: Towards Transactional Memory Support for GCC. In: *First International Workshop on GCC Research Opportunities*. GROW '09, January 2009 Held in conjunction with: the fourth International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC).

[173] Schindewolf, M., Esselson, A., Karl, W.: Compiler-Assisted Selection of a Software Transactional Memory System. In Berekovic, M., Fornaciari, W., Brinkschulte, U., Silvano, C., eds.: *Architecture of Computing Systems - ARCS 2011*, Volume 6566 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg 2011, Pages 147–157.

[174] Schindewolf, M., Karl, W.: Investigating Compiler Support for Software Transactional Memory. In: *Proceedings of ACACES 2009 Poster Abstracts: Advanced Computer Architecture and Compilation for Embedded Systems*, Terrassa, Spain, Academia Press, Ghent, July 2009, Pages 89–92.

[175] Schindewolf, M., Karl, W.: Capturing Transactional Memory Application's Behavior – The Prerequisite for Performance Analysis. In: *International Conference on Multicore Software Engineering, Performance and Tools (MSEPT 2012)*, Volume 7303 of *Lecture Notes in Computer Science*, Springer Verlag May 31–June 1, 2012, Pages 30–41, ISBN 978-3-642-31201-4.

[176] Schindewolf, M., Rocker, B., Karl, W., Heuveline, V.: Evaluation of two Formulations of the Conjugate Gradients Method with Transactional Memory. In Wolf, F., Mohr, B., an Mey, D., eds.: *19th International European Conference on Parallel and Distributed Computing Euro-Par 2013*, Volume 8097 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg 2013 Accepted for publication.

[177] Schindewolf, M., Schulz, M., Bihari, B., Gyllenhaal, J., Wang, A., Karl, W.: Performance Analysis of and Tool Support for Transactional Memory on BG/Q. In:

Euro-TM Workshop on Transactional Memory (WTM 2012), April 2012 Abstract available at `http://www.eurotm.org/action-meetings/wtm2012/program/abstracts#Schindewolf`.

[178] Schindewolf, M., Tao, J., Karl, W., Cintra, M.: A Generic Tool Supporting Cache Designs and Optimisation on Shared Memory Systems. In: *Proceedings of the 9th Workshop on Parallel Systems and Algorithms (PASA '08)*, Volume 124 of *Lecture Notes in Informatics (LNI)*, GI e.V. February 2008, Pages 69–79, ISBN 978-3-88579-218-5.

[179] Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. *Scientific Programming*, Volume 16, Number 2-3, 2008, IOS Press, Pages 105–121, ISSN 1058-9244.

[180] Shavit, N., Touitou, D.: Software Transactional Memory. In: *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. PODC '95, New York, NY, USA, ACM, 1995, Pages 204–213, ISBN 0-89791-710-3.

[181] Shen, X., Zhong, Y., Ding, C.: Locality Phase Prediction. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS-XI, New York, NY, USA, ACM, 2004, Pages 165–176, ISBN 1-58113-804-0.

[182] Shende, S., Malony, A., Morris, A., Wolf, F.: Performance Profiling Overhead Compensation for MPI Programs. In Di Martino, B., Kranzlmüller, D., Dongarra, J., eds.: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Volume 3666 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg 2005, Pages 359–367, ISBN 978-3-540-29009-4.

[183] Shende, S.S., Malony, A.D.: The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, Volume 20, Number 2, May 2006, Sage Publications, Inc., Pages 287–311, ISSN 1094-3420.

[184] Sherwood, T., Sair, S., Calder, B.: Phase Tracking and Prediction. *ACM SIGARCH Computer Architecture News*, Volume 31, Number 2, May 2003, ACM, Pages 336–349, ISSN 0163-5964.

[185] Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing Isolation and Ordering in STM. *ACM SIGPLAN Notices*, Volume 42, Number 6, June 2007, ACM, Pages 78–88, ISSN 0362-1340.

[186] Shriraman, A., Dwarkadas, S.: Refereeing Conflicts in Hardware Transactional Memory. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09, New York, NY, USA, ACM, 2009, Pages 136–146, ISBN 978-1-60558-498-0.

[187] Skodras, A., Christopoulos, C., Ebrahimi, T.: The JPEG 2000 Still Image Compression Standard. *Signal Processing Magazine, IEEE*, Volume 18, Number 5, sep 2001, Pages 36 –58, ISSN 1053-5888.

[188] Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with Isolation and Cooperation. *ACM SIGPLAN Notices*, Volume 42, Number 10, October 2007, ACM, Pages 191–210, ISSN 0362-1340.

[189] Sonmez, N., Arcas, O., Kirchhofer, P., Schindewolf, M., Unsal, O.S., Cristal, A., Karl, W.: A low-overhead Profiling and Visualization Framework for Hybrid Transactional Memory. In: *FCCM 2012: The 20th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2012, Pages 1–8 http://fccm12.cse.sc.edu/4699a001.pdf.

[190] Sonmez, N., Arcas, O., Pflucker, O., Unsal, O.S., Cristal, A., Hur, I., Singh, S., Valero, M.: TMbox: A Flexible and Reconfigurable 16-Core Hybrid Transactional Memory System. In: *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. FCCM '11, Washington, DC, USA, IEEE Computer Society, 2011, Pages 146–153, ISBN 978-0-7695-4301-7.

[191] Sonmez, N., Cristal, A., Unsal, O., Harris, T., Valero, M.: Profiling Transactional Memory Applications on an atomic block basis: A Haskell case study. In: *MULTIPROG 2009*, January 2009.

[192] Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A Comprehensive Strategy for Contention Management in Software Transactional Memory. *ACM SIGPLAN Notices*, Volume 44, Number 4, February 2009, ACM, Pages 141–150, ISSN 0362-1340.

[193] Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization Techniques for Software Transactional Memory. In: *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*. PODC '07, New York, NY, USA, ACM, 2007, Pages 338–339, ISBN 978-1-59593-616-5.

[194] Spear, M.F., Marathe, V.J., Scherer, W.N., Scott, M.L.: Conflict Detection and Validation Strategies for Software Transactional Memory. In: *Proceedings of the 20th International Conference on Distributed Computing*. DISC'06, Berlin, Heidelberg, Springer-Verlag, 2006, Pages 179–193, ISBN 3-540-44624-9, 978-3-540-44624-8.

[195] Spear, M.F., Silverman, M., Dalessandro, L., Michael, M.M., Scott, M.L.: Implementing and Exploiting Inevitability in Software Transactional Memory. *International Conference on Parallel Processing*, Volume 0, 2008, IEEE Computer Society, Pages 59–66, ISSN 0190-3918.

[196] Stollnitz, E.J., DeRose, T.D., Salesin, D.H.: Wavelets for Computer Graphics: A Primer, Part 1. *IEEE Computer Graphics and Applications*, Volume 15, 1995, IEEE Computer Society, Pages 76–84, ISSN 0272-1716.

[197] Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In: *IEEE Proceedings on Field-Programmable Custom Computing Machines*. FCCM '06, IEEE Computer Society Press May 2006, Pages 259–268 doi: 10.1109/FCCM.2006.57.

[198] Tallent, N.R., Mellor-Crummey, J., Franco, M., Landrum, R., Adhianto, L.: Scalable Fine-grained Call Path Tracing. In: *Proceedings of the International Conference on Supercomputing*. ICS '11, New York, NY, USA, ACM, 2011, Pages 63–74, ISBN 978-1-4503-0102-2.

[199] Tao, J., Gaugler, T., Karl, W.: A Profiling Tool for Detecting Cache-Critical Data Structures. In Kermarrec, A.M., Bougé, L., Priol, T., eds.: *Euro-Par 2007 Parallel*

Processing, Volume 4641 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg 2007, Pages 52–61, ISBN 978-3-540-74465-8.

[200] Wagner, R.S., Baraniuk, R.G., Du, S., Johnson, D.B., Cohen, A.: An Architecture for Distributed Wavelet Analysis and Processing in Sensor Networks. In: *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*. IPSN '06, New York, NY, USA, ACM, 2006, Pages 243–250, ISBN 1-59593-334-4.

[201] Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12, New York, NY, USA, ACM, 2012, Pages 127–136, ISBN 978-1-4503-1182-3.

[202] Wang, C., Chen, W.Y., Wu, Y., Saha, B., Adl-Tabatabai, A.R.: Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '07, Washington, DC, USA, IEEE Computer Society, 2007, Pages 34–48, ISBN 0-7695-2764-7.

[203] Watson, I., Kirkham, C., Lujan, M.: A Study of a Transactional Parallel Routing Algorithm. In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07, Washington, DC, USA, IEEE Computer Society, 2007, Pages 388–398, ISBN 0-7695-2944-5.

[204] Wee, S., Casper, J., Njoroge, N., Tesylar, Y., Ge, D., Kozyrakis, C., Olukotun, K.: A practical FPGA-based Framework for Novel CMP Research. In: *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. FPGA '07, New York, NY, USA, ACM, 2007, Pages 116–125, ISBN 978-1-59593-600-4.

[205] Welc, A., Hosking, A.L., Jagannathan, S.: Transparently Reconciling Transactions with Locking for Java Synchronization. In: *Proceedings of the 20th European Conference on Object-Oriented Programming*. ECOOP'06, Berlin, Heidelberg, Springer-Verlag, 2006, Pages 148–173, ISBN 3-540-35726-2, 978-3-540-35726-1.

[206] Welc, A., Saha, B., Adl-Tabatabai, A.R.: Irrevocable Transactions and their Applications. In: *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '08, New York, NY, USA, ACM, 2008, Pages 285–296, ISBN 978-1-59593-973-9.

[207] Wolf, F., Mohr, B.: EPILOG Binary Trace-Data Format. Technical report, Forschungszentrum Jülich, University of Tennessee May 2004 FZJ-ZAM-IB-2004-06.

[208] Wong, M., Bihari, B.L., de Supinski, B.R., Wu, P., Michael, M., Liu, Y., Chen, W.: A Case for Including Transactions in OpenMP. In: *IWOMP 2010 Conference Proceedings*, Tsukuba, Japan, LNCS 6132, June 2010, Pages 149–160.

[209] Wong, M., Bihari, B., de Supinski, B., Wu, P., Michael, M., Liu, Y., Chen, W.: A Case for Including Transactions in OpenMP. In Sato, M., Hanawa, T., Müller, M., Chapman, B., de Supinski, B., eds.: *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, Volume 6132 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg 2010, Pages 149–160, ISBN 978-3-642-13216-2.

[210] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA '95, New York, NY, USA, ACM Press, 1995, Pages 24–36, ISBN 0-89791-698-0.

[211] Wu, P., Michael, M.M., von Praun, C., Nakaike, T., Bordawekar, R., Cain, H.W., Cascaval, C., Chatterjee, S., Chiras, S., Hou, R., Mergen, M., Shen, X., Spear, M.F., Wang, H.Y., Wang, K.: Compiler and Runtime Techniques for Software Transactional Memory Optimization. *Concurrency and Computation: Practice & Experience*, Volume 21, Number 1, January 2009, John Wiley and Sons Ltd., Pages 7–23, ISSN 1532-0626.

[212] Yen, L., Bobba, J., Marty, M.R., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In: *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*. HPCA '07, Washington, DC, USA, IEEE Computer Society, 2007, Pages 261–272, ISBN 1-4244-0804-0.

[213] Yin, R.: *Case Study Research: Design and Methods*. 3rd edition Applied Social Research Methods Series, Sage Publications, 2002, ISBN 9780761925521.

[214] Yoo, R.M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.R., Lee, H.H.S.: Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In: *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '08, New York, NY, USA, ACM, 2008, Pages 265–274, ISBN 978-1-59593-973-9.

[215] Zannier, C., Melnik, G., Maurer, F.: On the Success of Empirical Studies in the International Conference on Software Engineering. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06, New York, NY, USA, ACM, 2006, Pages 341–350, ISBN 1-59593-375-1.

[216] Zhang, X., Wang, Z., Gloy, N., Chen, J.B., Smith, M.D.: System Support for Automatic Profiling and Optimization. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. SOSP '97, New York, NY, USA, ACM, 1997, Pages 15–26, ISBN 0-89791-916-5.

[217] Zilles, C.B., Sohi, G.S.: A Programmable Co-processor for Profiling. In: *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. HPCA '01, Washington, DC, USA, IEEE Computer Society, 2001, Pages 241–, ISBN 0-7695-1019-1.

[218] Zyulkyarov, F., Cristal, A., Cvijic, S., Ayguade, E., Valero, M., Unsal, O., Harris, T.: WormBench: A Configurable Workload for Evaluating Transactional Memory Systems. In: *Proceedings of the 9th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*. MEDEA '08, New York, NY, USA, ACM, 2008, Pages 61–68, ISBN 978-1-60558-243-6.

[219] Zyulkyarov, F., Gajinov, V., Unsal, O.S., Cristal, A., Ayguadé, E., Harris, T., Valero, M.: Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. *ACM SIGPLAN Notices*, Volume 44, Number 4, February 2009, ACM, Pages 25–34, ISSN 0362-1340.

[220] Zyulkyarov, F., Stipic, S., Harris, T., Unsal, O.S., Cristal, A., Hur, I., Valero, M.: Discovering and Understanding Performance Bottlenecks in Transactional Applications. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10, New York, NY, USA, ACM, 2010, Pages 285–294, ISBN 978-1-4503-0178-7.

Curriculum Vitae

Personal Information

Date of Birth:	01. April 1981
Place of Birth:	Eschwege, Germany
Citizenship:	German

Education

Since 04/2008		Research assistant with Chair for Computer Architecture and Parallel Processing of Prof. Wolfgang Karl and pursuing a PhD at the Karlsruhe Institute of Technology (KIT)
10/2001 to	02/2008	Study of informatics at University of Karlsruhe
	10/2003	Pre-degree, grade 2.0
	02/2008	Diploma with grade 1.2 (equivalent to Master in Computer Science with a six months thesis)
Electives:		Cryptography Embedded Systems Design and Computer Architecture
Seminar:		Supercomputers
Study Thesis:		Analysis of Cache Misses Using SIMICS
Diploma Thesis:		Development of a Compiler-based Validation Infrastructure Supporting Incremental Performance Model Analysis
Supervisor:		Prof. Wolfgang Karl, University of Karlsruhe
09/1997 to	06/2000	Oberstufengymnasium Eschwege German “Abitur” qualifying for university studies, grade 1.8

Studies and Research Abroad

09/2012 to 11/2012	Internship at the Lawrence Livermore National Laboratory (LLNL), Livermore, CA, US Joint work with Dr. Martin Schulz on tools for Transactional Memory
08/2011 to 12/2011	Internship at the Lawrence Livermore National Laboratory (LLNL), Livermore, CA, US Working with Dr. Martin Schulz on performance of Transactional Memory on the BG/Q architecture
06/2008	Institut National de Recherche en Informatique et en Automatique, INRIA Saclay - Île de France, Orsay cedex, France Joint work with Albert Cohen on Transactional Memory Support for GCC. Project funded by HiPEAC Network of Excellence
06/2007 to 08/2007	Institute for Computing Systems Architecture, University of Edinburgh, UK Diploma Thesis: Development of a Compiler-based Validation Infrastructure Supporting Incremental Performance Model Analysis
03/2006 to 04/2006	Institute for Computing Systems Architecture, University of Edinburgh, UK Study Thesis: Analysis of Cache Misses Using SIMICS

Practical Experience

10/2003 to 09/2006	Institute of Analysis, University of Karlsruhe Administration of Computer Systems and Networks
08/2000 to 06/2001	“Zentrum für soziale Psychiatrie Werra-Meißner”, Hessisch Lichtenau, Germany Alternative Civilian Service (in a hospital instead of Military Service)