Karlsruhe Institute of Technology

Proactive Adaptation in Self-Organizing Task-based Runtime Systems for Different Computing Classes

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

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

genehmigte

Dissertation

von

Thomas Becker
aus Karlsruhe

Tag der mündlichen Prüfung:	25. Mai 2022
Erster Gutachter:	Prof. Dr. rer. nat. Wolfgang Karl
Zweiter Gutachter:	Prof. Dr. rer. nat. Martin Schulz

Abstract

Modern computing systems offer a high degree of parallelism and heterogeneity to users and application developers. However, efficiently utilizing these systems requires deep knowledge, e.g., of the underlying hardware platform and different programming models, and extensive work from a developer. In this thesis, efficient usage is related to application makespan, the system's energy consumption, the maximum temperature of processing units, and system reliability. Next to multiple optimization goals, an application developer has to consider the system's specific constraints like application deadlines or safety guarantees that come with certain fields of application. This complexity of heterogeneous systems makes it impossible to predict all potential system states and environmental effects that could occur at runtime. Therefore, the system and application developers are not able to determine at design time how the system and applications should react in all potential situations. For this reason, systems have to be dynamically adapted at runtime to optimize their behavior according to the current situation. E.g., in embedded systems with limited cooling capacities, load balancing, frequency reduction, or switching off processing units to reduce heat is necessary when a certain temperature threshold is reached.

But just reacting to a disadvantageous system state is usually not enough. The objective should be to proactively avoid disadvantageous or faulty system states altogether to lessen the necessity to call emergency functionality and improve the user experience. For example instead of reducing heat by rebalancing tasks, proactive mechanisms could avoid critical temperatures beforehand by delaying certain uncritical tasks or reducing their accuracy or quality of service (QoS). Thereby, the system load gets reduced before a critical point is reached.

Solutions offered by the literature like uniform programming languages or runtime systems address some of the aforementioned challenges, however no approach exists that is able to dynamically and, in particular, proactively balance multiple contradicting optimization goals. A concept that unburdens the developer from this complexity and provides a way to dynamically and proactively adapt to changes is self-organization. However, self-organization is defined as a process without external control or guidance. In the context of system optimization this can easily lead to undesired results. An approach that combines self-organization with a control mechanism that aims for robustness and resilience against outside disturbances is organic computing. The defining characteristic of organic computing is an observer/controller architecture. The concept of this architecture is monitoring the current state of the system and the environment, analyzing this data, and making decisions about the future system behavior based on this analysis. I.e., organic computing enables to proactively select and trigger mechanisms that optimize the system and avoid undesired states based on past and current states.

To transfer the benefits of organic computing to modern heterogeneous systems, I combine the organic computing approach with a runtime system. Runtime systems are a promising candidate to implement the organic computing approach as they already

monitor and control the execution of applications. In particular, I work on the following research topics in this thesis by combining the concepts of organic computing and runtime systems:

- Capturing the current system state by monitoring sensors and performance counters
- Predicting future system states by analyzing past behavior
- Utilizing state information to proactively adapt the system

I provide extensions to the topic of system state capturing in two ways. First, I introduce a *novel heuristic metric for a processing unit's reliability* based on *symptom-based fault detection*. Symptom based fault detection is a light-weight method to dynamically detect soft hardware errors by monitoring execution behavior with performance counters. Dynamically detecting faults then allows to compute a heuristic fault rate for a processing unit in a specific time window. The fault rate is employed to compute the number of required executions of a task to guarantee a given result reliability. An important aspect of system state capturing is minimizing the emerging overhead. I reduce the number of necessary profiling runs for OpenMP tasks via thread interpolation and scaling checks. Additionally, I study *predicting OpenCL task execution times* without executing them. The prediction models are trained with different machine learning algorithms. Profiles of the kernels created by static code analysis are thereby used as input.

To forecast future system states, I focus on *predicting upcoming tasks/applications* that will arrive in the system in the near future. Combined with the monitoring database, this enables to estimate the upcoming costs that the system has to handle. This thesis creates two task prediction mechanisms, one targeting independent tasks that repeatedly create new task instances and one targeting dependent applications that form execution patterns. Both mechanisms deploy a prediction table based on Markov predictors and pattern matching.

In this thesis, the knowledge that is acquired through system monitoring and task prediction is used to proactively *balance the system's optimization goals*. This is done through a set of rules that maps a system state description, consisting of the current state, predictions, and system constraints, onto a set of weights. To learn these rules, an *extended classifier system XCS* is employed. The XCS is embedded in a hierarchical architecture designed by organic computing principles. Hereby, an important design decision is outsourcing the creation of new rules to an offline algorithm that utilizes simulation and runs in parallel to the normal tasks of the system. So, no untested rules whose effects are not yet known are added to the live system. The resulting weights are then used to build an evaluation function for list scheduling algorithms.

This thesis adds to the research field task scheduling algorithms by providing two extensions to dynamic scheduling algorithms. The first extension focuses on non-safety-critical systems that utilize priorities to express differing application importance. As static

priorities may lead to starvation in highly utilized systems, I created a *dynamic aging mechanism* that is able to adapt task priorities according to the current utilization and task waiting times. Thereby, the mechanism reduces the total makespan over all tasks and the waiting time for lower priority tasks.

As of yet, a great number of applications is not ready to capitalize on the high degree of parallelism offered by modern computing systems. A concept that tries to solve this problem by scheduling several different processes on the same computing node is co-scheduling. In this thesis, I introduce a *novel co-scheduling mechanism* that is able to optimize the task schedules of several runtime system instances executing on the same computing node. To share the necessary information between the runtime system instances, the mechanism stores the data in shared memory. When a runtime system inserts new tasks into the system, the mechanism checks if computing a new schedule is sensible. If the decision to create a new schedule is made, the mechanism employs *simulated annealing* to schedule all tasks which have not yet started their execution.

To summarize, this thesis offers novel mechanisms and algorithms, and extensions to several research fields in order to implement a proactive self-organizing system that is able to adapt to new and unknown situations. Thereby, complexity for users and application developers is reduced by outsourcing decision making into the system itself. Simultaneously, this approach maintains the efficient usage of the system's resources. In total, this thesis makes the following contributions:

- Introducing a novel heuristic metric to measure reliability of processing units. The metric is based on the light-weight fault detection method symptom-based fault detection. Symptom-based fault detection is able to reliably detect several injected fault classes and interferences simulating soft hardware errors on both a CPU and a GPU. Furthermore, it confirms these results statistically in Welch's t-test.

- Proposing an OpenCL kernel execution time prediction model based on static code analysis. The model is able to select the fastest processing unit out of a set of processing units with an accuracy of $69\,\%$ in the worst case compared to a baseline accuracy of $25\,\%$ that always predicts the processor that dominates the highest amount of kernels. In the best case, the model achieves an accuracy of up to $83\,\%$.

- Providing two prediction mechanisms for upcoming tasks, one targeting independent tasks that constantly create new task instances and one targeting dependent applications that form execution patterns. The first mechanism obtains an maximum $sMAPE$ value of $4.33\,\%$ for sporadic and $0.002\,\%$ for periodic tasks while predicting the time period between two consecutive task instances. Additionally, it reliably detects tasks that posses an aperiodic execution scheme. The second predictor achieves an accuracy of $77.6\,\%$ forecasting the next upcoming task and its starting time.

- Introducing an implementation of a hierarchical organic computing framework including a modified XCS in the context of task scheduling. To implement the framework, a novel reward mechanism utilizing a specifically implemented task execution cost simulator is proposed. The XCS maps system state descriptions onto weights to balance the system's optimization objectives. Integrated as an evaluation function for a list scheduling algorithm, this concepts on average decreases the makespan by $10.4\,\%$ or $26.7\,s$, the energy consumption by $4.7\,\%$ or $2061.1\,J$, and the maximum temperature of the GPU by $3.6\,\%$ or $2.7K$ while only increasing the maximum CPU core temperature by $6\,\%$ or $2.3\,K$ in an evaluation scenario consisting of an application pattern that is repeated five times.

- Proposing two extensions that improve dynamic task scheduling for a single and multiple processes, e.g., multiple runtime system instances. The first mechanism, an aging algorithm, targets non-safety critical systems that utilize task priorities to represent differing application importance. To avoid starvation in such scenarios, it dynamically adapts task priorities according to the current utilization and task waiting times. Overall, this mechanism achieves an average speed up of $3.75\,\%$ and $3.16\,\%$ in two evaluation scenarios while reducing the flow time of lower priority tasks by up to $25,67\,\%$. The second mechanism enables the schedule optimization of several runtime system instances executing on the same computing node in parallel. This co-scheduling approach employs shared memory to share information between the processes and simulated annealing to compute new schedules. In two evaluation scenarios the mechanism achieves average speed ups of $19.74\,\%$ and $20.91\,\%$ or about $2.7\,s$ and $3\,s$, respectively.

Zusammenfassung

Moderne Computersysteme bieten Anwendern und Anwendungsentwicklern ein hohes Maß an Parallelität und Heterogenität. Die effiziente Nutzung dieser Systeme erfordert jedoch tiefgreifende Kenntnisse, z.B. der darunterliegenden Hardware-Plattform und den notwendigen Programmiermodellen, und umfangreiche Arbeit des Entwicklers. In dieser Thesis bezieht sich die effiziente Nutzung auf die Gesamtausführungszeit der Anwendungen, den Energieverbrauch des Systems, die maximale Temperatur der Verarbeitungseinheiten und die Zuverlässigkeit des Systems. Neben den verschiedenen Optimierungszielen muss ein Anwendungsentwickler auch die spezifischen Einschränkungen und Randbedingungen des Systems berücksichtigen, wie z. B. Deadlines oder Sicherheitsgarantien, die mit bestimmten Anwendungsbereichen einhergehen. Diese Komplexität heterogener Systeme macht es unmöglich, alle potenziellen Systemzustände und Umwelteinflüsse, die zur Laufzeit auftreten können, vorherzusagen. Die System- und Anwendungsentwickler sind somit nicht in der Lage, zur Entwurfszeit festzulegen, wie das System und die Anwendungen in allen möglichen Situationen reagieren sollen. Daher ist es notwendig, die Systeme zur Laufzeit der aktuellen Situation anzupassen, um ihr Verhalten entsprechend zu optimieren. In eingebetteten Systemen mit begrenzten Kühlkapazitäten muss z.B. bei Erreichen einer bestimmten Temperaturschwelle eine Lastverteilung vorgenommen, die Frequenz verringert oder Verarbeitungseinheiten abgeschaltet werden, um die Wärmeentwicklung zu reduzieren.

Normalerweise reicht es aber nicht aus, einfach nur auf einen ungünstigen Systemzustand zu reagieren. Das Ziel sollte darin bestehen, ungünstige oder fehlerhafte Systemzustände vor dem Auftreten zu vermeiden, um die Notwendigkeit des Aufrufs von Notfallfunktionen zu verringern und die Benutzerfreundlichkeit zu verbessern. Anstatt beispielsweise die Wärmeentwicklung durch eine Neuverteilung der Anwendungen zu reduzieren, könnten proaktive Mechanismen kritische Temperaturen bereits im Vorfeld vermeiden, indem sie bestimmte unkritische Aufgaben verzögern oder deren Genauigkeit oder QoS verringern. Auf diese Weise wird die Systemlast reduziert, bevor ein kritischer Punkt erreicht wird.

Lösungen des aktuellen Stands der Technik wie einheitliche Programmiersprachen oder Laufzeitsysteme adressieren einige der oben genannten Herausforderungen, jedoch existiert kein Ansatz, der in der Lage ist, eine Optimierung mehrerer sich widersprechender Zielfunktionen dynamisch und vor allem proaktiv durchzuführen. Ein Konzept, das diese komplexe Aufgabe für den Entwickler übernimmt und eine Möglichkeit zur dynamischen und proaktiven Anpassung an Veränderungen bietet, ist die Selbstorganisation. Selbstorganisation ist jedoch definiert als ein Prozess ohne externe Kontrolle oder Steuerung. Im Kontext der Systemoptimierung kann dies leicht zu unerwünschten Ergebnissen führen. Ein Ansatz, der Selbstorganisation mit einem Kontrollmechanismus

kombiniert, welcher auf Robustheit und Widerstandsfähigkeit gegenüber äußeren Störungen abzielt, ist Organic Computing. Das bestimmende Merkmal von Organic Computing ist eine Observer/Controller-Architektur. Das Konzept dieser Architektur besteht darin, den aktuellen Zustand des Systems und der Umgebung zu überwachen, diese Daten zu analysieren und auf der Grundlage dieser Analyse Entscheidungen über das zukünftige Systemverhalten zu treffen. Organic Computing ermöglicht es also auf der Grundlage der vergangenen und des aktuellen Zustands proaktiv Mechanismen auszuwählen und auszulösen, die das System optimieren und unerwünschte Zustände vermeiden.

Um die Vorteile des Organic Computings auf moderne heterogene Systeme zu übertragen, kombiniere ich den Organic Computing-Ansatz mit einem Laufzeitsystem. Laufzeitsysteme sind ein vielversprechender Kandidat für die Umsetzung des Organic Computing-Ansatzes, da sie bereits die Ausführung von Anwendungen überwachen und steuern. Insbesondere betrachte und bearbeite ich in dieser Dissertation die folgenden Forschungsthemen, indem ich die Konzepte des Organic Computings und der Laufzeitsysteme kombiniere:

- Erfassen des aktuellen Systemzustands durch Überwachung von Sensoren und Performance Countern
- Vorhersage zukünftiger Systemzustände durch Analyse des vergangenen Verhaltens
- Nutzung von Zustandsinformationen zur proaktiven Anpassung des Systems

Ich erweitere das Thema der Erfassung von Systemzuständen auf zwei Arten. Zunächst führe ich eine *neuartige heuristische Metrik zur Berechnung der Zuverlässigkeit einer Verarbeitungseinheit* ein, die auf *symptombasierter Fehlererkennung* basiert. Symptombasierte Fehlererkennung ist eine leichtgewichtige Methode zur dynamischen Erkennung von soften Hardware-Fehlern durch Überwachung des Ausführungsverhaltens mit Performance Countern. Die dynamische Erkennung von Fehlern ermöglicht dann die Berechnung einer heuristischen Fehlerrate einer Verarbeitungseinheit in einem bestimmten Zeitfenster. Die Fehlerrate wird verwendet, um die Anzahl der erforderlichen Ausführungen einer Anwendung zu berechnen, um eine bestimmte Ergebniszuverlässigkeit, also eine Mindestwahrscheinlichkeit für ein korrektes Ergebnis, zu gewährleisten. Ein wichtiger Aspekt der Zustandserfassung ist die Minimierung des entstehenden Overheads. Ich verringere die Anzahl der für OpenMP-Tasks notwendigen Profiling-Durchläufe durch Thread-Interpolation und Überprüfungen des Skalierungsverhaltens. Zusätzlich untersuche ich die Vorhersage von OpenCL Task-Ausführungszeiten. Die Prädiktoren der Ausführungszeiten werden mit verschiedenen maschinellen Lernalgorithmen trainiert. Als Input werden Profile der Kernel verwendet, die durch statische Codeanalyse erstellt wurden.

Um in dieser Dissertation zukünftige Systemzustände vorherzusagen, sollen Anwendungen vorausgesagt werden, die in naher Zukunft im System vorkommen werden. In

Kombination mit der Ausführungsdatenbank ermöglicht dies die Schätzung der anstehenden Kosten, die das System zu bewältigen hat. In dieser Arbeit werden zwei Mechanismen zur Vorhersage von Anwendungen/Tasks entwickelt. Der erste Prädiktor zielt darauf ab, neue Instanzen unabhängiger Tasks vorherzusagen. Der zweite Mechanismus betrachtet Ausführungsmuster abhängiger Anwendungen und sagt auf dieser Grundlage zukünftig auftretende Anwendungen vorher. Beide Mechanismen verwenden eine Vorhersagetabelle, die auf Markov-Prädiktoren und dem Abgleich von Mustern basiert.

In dieser Arbeit wird das Wissen, das durch die Systemüberwachung und die Vorhersage zukünftiger Anwendungen gewonnen wird, verwendet, um die Optimierungsziele des Systems proaktiv in Einklang zu bringen und zu gewichten. Dies geschieht durch eine Reihe von Regeln, die eine Systemzustandsbeschreibung, bestehend aus dem aktuellen Zustand, Vorhersagen und Randbedingungen bzw. Beschränkungen, auf einen Vektor aus Gewichten abbilden. Zum Erlernen der Regelmenge wird ein *Extended Classifer System (XCS)* eingesetzt. Das XCS ist in eine hierarchische Architektur eingebettet, die nach den Prinzipien des Organic Computing entworfen wurde. Eine wichtige Designentscheidung ist dabei die Auslagerung der Erstellung neuer Regeln an einen Offline-Algorithmus, der einen Simulator nutzt und parallel zum normalen Systemablauf ausgeführt wird. Dadurch wird sichergestellt, dass keine ungetesteten Regeln, deren Auswirkungen noch nicht bekannt sind, dem laufenden System hinzugefügt werden. Die sich daraus ergebenden Gewichte werden schließlich verwendet, um eine Bewertungsfunktion für List Scheduling-Algorithmen zu erstellen.

Diese Dissertation erweitert das Forschungsgebiet der Scheduling-Algorithmen durch zwei Mechanismen für dynamisches Scheduling. Die erste Erweiterung konzentriert sich auf nicht sicherheitskritische Systeme, die Prioritäten verwenden, um die unterschiedliche Wichtigkeit von Tasks auszudrücken. Da statische Prioritäten in stark ausgelasteten Systemen zu Starvation führen können, habe ich einen dynamischen Ageing-Mechanismus entwickelt, der dazu in der Lage ist, die Prioritäten der Tasks entsprechend der aktuellen Auslastung und ihrer Wartezeiten anzupassen. Dadurch reduziert der Mechanismus die Gesamtlaufzeit über alle Tasks und die Wartezeit für Tasks mit niedrigerer Priorität.

Noch ist eine große Anzahl von Anwendungen nicht dazu bereit, den hohen Grad an Parallelität zu nutzen, den moderne Computersysteme bieten. Ein Konzept, das versucht dieses Problem zu lösen, indem es mehrere verschiedene Prozesse auf demselben Rechenknoten zur Ausführung bringt, ist das Co-Scheduling. In dieser Dissertation stelle ich einen neuartigen Co-Scheduling-Mechanismus vor, welcher die Task-Schedules mehrerer Laufzeitsysteminstanzen optimiert, die auf demselben Rechenknoten ausgeführt werden. Um die notwendigen Informationen zwischen den Laufzeitsysteminstanzen zu teilen, speichert der Mechanismus die Daten in Shared Memory. Sobald ein Laufzeitsystem neue Tasks in das System einfügt, prüft der Mechanismus, ob die Berechnung eines neuen Schedules sinnvoll ist. Wird die Entscheidung getroffen, einen neuen Sche-

dule zu berechnen, setzt der Mechanismus *Simulated Annealing* ein, um alle Tasks, die bisher noch nicht mit ihrer Ausführung begonnen haben, neu auf Ausführungseinheiten abzubilden.

Zusammenfassend lässt sich sagen, dass diese Arbeit neuartige Mechanismen und Algorithmen sowie Erweiterungen zu verschiedenen Forschungsgebieten anbietet, um ein proaktives selbst-organisierendes System zu implementieren, das sich an neue und unbekannte Situationen anpassen kann. Dabei wird die Komplexität für Benutzer und Anwendungsentwickler reduziert, indem die Entscheidungsfindung in das System selbst ausgelagert wird. Gleichzeitig sorgt dieser Ansatz für eine effiziente Nutzung der Ressourcen des Systems. Insgesamt leistet diese Arbeit die folgenden Beiträge zur Erweiterung des Stands der Forschung:

- Einführung einer neuartigen heuristischen Metrik zur Messung der Zuverlässigkeit von Verarbeitungseinheiten. Die Metrik basiert auf einer leichtgewichtigen Methode zur Fehlererkennung, genannt symptombasierte Fehlererkennung. Mit der symptombasierten Fehlererkennung ist es möglich, mehrere injizierte Fehlerklassen und Interferenzen, die Soft-Hardware-Fehler simulieren, sowohl auf einer CPU als auch auf einer GPU zuverlässig zu erkennen. Darüber hinaus werden diese Ergebnisse durch Welch's t-Test statistisch bestätigt.

- Vorschlag eines Vorhersagemodells für die Ausführungszeit von OpenCL Kerneln, das auf statischer Code-Analyse basiert. Das Modell ist in der Lage, die schnellste Verarbeitungseinheit aus einer Menge von Verarbeitungseinheiten mit einer Genauigkeit von im schlechtesten Fall $69\,\%$ auszuwählen. Zum Vergleich: eine Referenzvariante, welche immer den Prozessor vorhersagt, der die meisten Kernel am schnellsten ausführt, erzielt eine Genauigkeit von $25\,\%$. Im besten Fall erreicht das Modell eine Genauigkeit von bis zu $83\,\%$.

- Bereitstellung von zwei Prädiktoren für kommende Tasks/Anwendungen. Der erste Mechanismus betrachtet unabhängige Tasks, die ständig neue Task-Instanzen erstellen, der zweite abhängige Anwendungen, die Ausführungsmuster bilden. Dabei erzielt der erste Mechanismus bei der Vorhersage der Zeitspanne zwischen zwei aufeinanderfolgenden Task-Instanzen einen maximalen $sMAPE$-Wert von $4,33\,\%$ für sporadische und $0,002\,\%$ für periodische Tasks. Darüber hinaus werden Tasks mit einem aperiodischen Ausführungsschema zuverlässig erkannt. Der zweite Mechanismus erreicht eine Genauigkeit von $77,6\,\%$ für die Vorhersage der nächsten anstehenden Anwendung und deren Startzeit.

- Einführung einer Umsetzung eines hierarchischen Organic Computing Frameworks mit dem Anwendungsgebiet Task-Scheduling. Dieses Framework enthält u.a. ein modifiziertes XCS, für dessen Design und Implementierung ein neuartiger Reward-Mechanismus entwickelt wird. Der Mechanismus bedient sich dabei eines speziell

für diesen Zweck entwickelten Simulators zur Berechnung von Task-Ausführungskosten. Das XCS bildet Beschreibungen des Systemzustands auf Gewichte zur Balancierung der Optimierungsziele des Systems ab. Diese Gewichte werden in einer Bewertungsfunktion für List Scheduling-Algorithmen verwendet. Damit wird in einem Evaluationsszenario, welches aus einem fünfmal wiederholten Muster aus Anwendungen besteht, eine Reduzierung der Gesamtlaufzeit um $10,4\,\%$ bzw. $26,7\,s$, des Energieverbrauchs um $4,7\,\%$ bzw. $2061,1\,J$ und der maximalen Temperatur der GPU um $3,6\,\%$ bzw. $2,7K$ erzielt. Lediglich die maximale Temperatur über alle CPU-Kerne erhöht sich um $6\,\%$ bzw. $2,3\,K$.

- Entwicklung von zwei Erweiterungen zur Verbesserung des dynamischen Task-Schedulings für einzelne und mehrere Prozesse, z.B. mehrere Laufzeitsysteminstanzen. Der erste Mechanismus, ein Ageing-Algorithmus, betrachtet nicht sicherheitskritische Systeme, welche Task-Prioritäten verwenden, um die unterschiedliche Bedeutung von Anwendungen darzustellen. Da es in solchen Anwendungsszenarien in Kombination mit hoher Systemauslastung zu Starvation kommen kann, passt der Mechanismus die Task-Prioritäten dynamisch an die aktuelle Auslastung und die Task-Wartezeiten an. Insgesamt erreicht dieser Mechanismus in zwei Bewertungsszenarien eine durchschnittliche Laufzeitverbesserung von $3,75\,\%$ und $3,16\,\%$ bei gleichzeitiger Reduzierung der Durchlaufzeit von Tasks mit niedrigerer Priorität um bis zu $25,67\,\%$. Der zweite Mechanismus ermöglicht die Optimierung von Schedules mehrerer Laufzeitsysteminstanzen, die parallel auf demselben Rechenknoten ausgeführt werden. Dieser Co-Scheduling-Ansatz verwendet Shared Memory zum Austausch von Informationen zwischen den Prozessen und Simulated Annealing zur Berechnung neuer Task-Schedules. In zwei Evaluierungsszenarien erzielt der Mechanismus durchschnittliche Laufzeitverbesserungen von $19,74\,\%$ und $20,91\,\%$ bzw. etwa $2,7\,s$ und $3\,s$.

Acknowledgments

Writing a PhD thesis is a long and hard journey of many ups and downs. Successfully completing it requires a great amount of support on the way and I owe my deepest and heartfelt gratitude to these people.

First and foremost, I am extremely grateful to my Ph.D. adviser, Prof. Dr. Wolfgang Karl., for always offering support and feedback when needed. Not only did he provide guidance in finding an appropriate research topic for my work, he also taught me how to properly structure my thesis and present my results to the reader. Furthermore, Prof. Dr. Karl creates an enjoyable working experience with a great balance of freedom and structure that nurtures the creative freedom necessary to tackle years of research.

I would also like to offer my sincere thanks to my co-referee Prof. Dr. Martin Schulz. Without hesitation, he agreed to review my thesis and offered his time in an extremely tight work schedule. Before being my co-referee, I already had the pleasure to work with Prof. Dr. Martin Schulz in a research project. His valuable insights and expertise helped to make our project a success and thereby also very positively influenced my thesis.

Furthermore, I would like to express my gratitude to my current and former colleagues (in alphabetical order): Dr. Michael Bromberger, Markus Hoffmann, Manuel Kalmbach, Dr. Mario Kicherer, Roman Lehmann, Oliver Mattes, and Dr. Anas Toma. Over the years, they offered valuable help, feedback, great discussions and conversations far beyond the scope of our research. It was always a pleasure working with you. An additional special thanks to Dr. Mario Kicherer who designed and developed the runtime system HALadapt in its original form and therefore built the basis for this work. I would also like to thank my colleagues from the other chairs of our department, namely the Chair of Embedded Systems, the Chair of Dependable Nano Computing, and the Operating Systems Group. Additionally, I am thankful to the current and former staff of our department for their assistance over the years.

All of the projects that I had the pleasure to work in during the course of my thesis influenced and are a part of this work. Hence, I would like to express my gratitude to my project partners from JGU Mainz, RWTH Aachen, Siemens, and TU München for the flawless cooperation and the fruitful meetings and discussions. A thesis is not possible without students who contribute through writing theses or their effort as student assistants. Therefore, I would like to extend my sincere thanks to all the students I supervised.

Most importantly, I am extremely and eternally grateful to my mother Luzia who always supported me and was there whenever I needed her. This goes far beyond what can be expected of a parent and I could not have done my studies or this thesis without her help. I would also like to sincerely thank my two sisters Melissa and Claudia and my father Karl for their continuous support and love.

Special thanks to Kevin Huttinger, Roman Lehmann, and German Pustovojtovskij who offered their help and spent their free time reviewing this thesis. Without feedback a thesis is not possible and for that I am deeply grateful. Last, but not least I would like

to heartily thank all my friends without whom life would be empty and certainly way less fun. Countless hours of time doing all kinds of activities together have made the years spent writing this thesis much easier and way more enjoyable. Without your diversions, I most certainly would not have stayed sane.

Contents

List of Own Relevant Publications

[10] Thomas Becker, Wolfgang Karl, and Tobias Schüle. “Evaluating Dynamic Task Scheduling in a Task-Based Runtime System for Heterogeneous Architectures”. In: *Architecture of Computing Systems – ARCS 2019*. Ed. by Martin Schoeberl, Christian Hochberger, et al. Cham: Springer International Publishing, 2019, pp. 142–155. ISBN: 978-3-030-18656-2.

[11] Thomas Becker, Pablo Busse, and Tobias Schuele. “Evaluation of Dynamic Task Scheduling Algorithms in a Runtime System for Heterogeneous Architectures”. In: *PARS-Mitteilungen* 35.1 (2018).

[12] Thomas Becker and Tobias Schüle. “Evaluating Dynamic Task Scheduling with Priorities and Adaptive Aging in a Task-Based Runtime System”. In: *Architecture of Computing Systems – ARCS 2020*. Ed. by André Brinkmann, Wolfgang Karl, et al. Cham: Springer International Publishing, 2020, pp. 17–31. ISBN: 978-3-030-52794-5.

[13] Thomas Becker, Nico Rudolf, et al. “Symptom-based Fault Detection in Modern Computer Systems”. In: *PARS-Mitteilungen* 36.1 (2019).

[14] Markus Helwig and Thomas Becker. “Predicting Efficient Execution with Source Code Analysis in a Heterogeneous Environment”. In: *PARS-Mitteilungen* 34.1 (2017).

[15] Thomas Becker. “Integrating Organic Computing Mechanisms into a Task-based Runtime System for Heterogeneous Systems”. In: *INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik – Informatik für Gesellschaft (Workshop-Beiträge)*. Ed. by Claude Draude, Martin Lange, and Bernhard Sick. Bonn: Gesellschaft für Informatik e.V., 2019, pp. 531–544.

[16] Thomas Becker, Dai Yang, et al. “Co-Scheduling in a Task-Based Programming Model”. In: *Proceedings of the 3rd Workshop on Co-Scheduling of HPC Applications (COSH 2018)*. Ed. by Carsten Trinitis and Josef Weidendorfer. Manchester, United Kingdom, Jan. 2018, pp. 9–14.

List of Figures

List of Tables

Part I

Motivation and Approach

CHAPTER

ONE

INTRODUCTION

Modern computer architectures feature a high degree of parallelism and heterogeneity. Efficiently using a heterogeneous parallel architecture demands parallelizing applications. Additionally, the inclusion of different accelerators results in the need of using different programming models and deep platform knowledge for an application developer. The goal should be to hide this complexity, but still enable the efficient usage of resources concerning makespan, energy consumption, heat dissipation, and system reliability. Additionally, the computing class' constraints have to be considered. For example, reaching deadlines, minimizing energy consumption and heat dissipation, and safety constraints, e.g., guaranteeing operational safety in the presence of faults in automobiles, are of great importance in embedded systems, whereas in high-performance computing (HPC) and desktop computing the focus lies on maximizing throughput or minimizing the total makespan.

The complexity of heterogeneous systems makes it impossible to predict all potential system states and environmental effects that could occur at runtime. Therefore, the system and application developers are not able to determine at design time how the system and applications should react in all potential situations. So, besides adapting optimization goals to the computing class' constraints, it is also necessary to dynamically adapt them to the current and prospectively possible system states, application requirements, and environmental effects. E.g., in embedded systems with limited cooling capacities, load balancing, frequency reduction, or switching off processing units to reduce heat is necessary when a certain temperature threshold is reached. However, the applications running in the system and their most reasonable distribution at this point in time are not known at design time. Next to the load, the temperature could also be influenced by the temperature of the system surroundings. Thus, a simple correlation between the applications ready-to-execute and the resulting system temperature would not be possible. For HPC server systems, users constantly send new and unknown jobs that have to be executed while potentially sharing the system's computing resources with other jobs.

Another important aspect is proactively avoiding disadvantageous or faulty system states to lessen the necessity to call emergency functionality and improve the user experience, especially in embedded systems. Instead of reducing heat by rebalancing tasks, proactive mechanisms could avoid critical temperatures beforehand by delaying certain uncritical tasks or reducing their accuracy or quality of service. Thereby, the system load gets reduced before a critical point is reached.

In the literature, approaches to hide the complexity of heterogeneous systems exist. Open Computing Language (OpenCL) [1] offers a uniform programming model, but still needs detailed hardware knowledge and is not able to dynamically adapt to new situations. This is also true for programming concepts like OpenMP [2] and MPI [3], that support heterogeneous accelerators in their latest versions. SYCL [4] is a higher-level programming model inspired by OpenCL and developed by the Khronos Group to simplify the programming of heterogeneous architectures. The uniform programming language is based on C++ and offers unified shared memory and a wide range of C++ features that can be used by the developer to create a heterogeneous application. However, the developer has to manually decide on which device a kernel should be executed and support for dynamic adaptation is not provided.

A different solution is offered by task-based runtime systems like HALadapt [5] and StarPU [6]. They abstract application development from the underlying hardware via a library-based approach. Here, a user is able to define a specific functionality, e.g., a matrix multiplication, and provide different implementation versions. At runtime, these systems are then able to select a fitting pair of processing unit and implementation. However, the current state of the art mainly focuses on makespan minimization or only statically balances makespan and energy consumption.

A solution that combines a unified programming model with a runtime system approach that is supported by a hardware concept is provided by the Heterogeneous System Architecture (HSA) [7]. HSA offers queues for the computing devices of a system and allows work stealing between the queues to balance load. Though, HSA has to be supported by the underlying hardware but as of yet only a limited amount of AMD or ARM architectures offer HSA support and features. Additionally, simple work stealing scheduling is not able to dynamically adapt to changing and new situations where different optimization goals become important. There also has been no further updates to the HSA specification, new press releases or publications since 2018.

To overcome the challenges created by the deployment of heterogeneous systems in different situations, application scenarios, and computing classes, it is necessary to dynamically compromise between contradictory optimization goals and proactively optimize the system. Self-organization (SO) is a way to provide a solution to these challenges.

It is defined by Camazine et al. [8] as follows:
"*SO is a process in which pattern at the global level of a system emerges solely from numerous interaction among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern*".

However, a missing control mechanism can easily lead to undesired results. A concept that aims for controlled SO is Organic Computing (OC) [9]. Next to self-organisation, an important feature of OC is robustness, the ability of a system to become more resilient against disturbances and attacks from the outside [9]. To achieve these features, OC systems deploy an observer/controller architecture. The concept of this architecture is monitoring the current state of the system and the environment, analyzing this data, and making decisions about the future system behavior based on this analysis. In the context of the aforementioned challenges, monitoring and analyzing the current and past states enables to draw conclusions about possible future system states. This is the basis to proactively select and trigger mechanisms that optimize the system and avoid undesired states. In summary, OC enables the efficient usage of systems in different, dynamic computing classes like automotives or HPC servers because adapting to unknown system states gets possible.

Runtime systems already provide possibilities to capture system states by monitoring because they control task execution. Additionally, runtime systems enable dynamic adaptations of the system. Thus, integrating the concept of organic computing into existing runtime systems is a logical solution to the aforementioned challenges. A combination of runtime systems and organic computing concepts offers many interesting research directions and poses several questions that have yet to be explored. In total, this dissertation answers the following research challenges and provides several contributions and solutions:

- Analysis of task-based runtime systems in different use cases and computing classes of heterogeneous systems to find occurring requirements and optimization goals. Thereby, it is possible to derive sensible restrictions and configurations for a system to be created. The analysis shows the necessity for proactive, dynamic adaptation and therefore motivates the remainder of this thesis.

- A thorough investigation and analysis of methods and tools to capture and evaluate the system state by collecting monitoring data. Although much information has to be collected and processed, creating as little overhead as possible is of great importance. Thereby, a heuristic reliability metric based on the light-weight concept symptom-based fault detection is created to extend the profiling possibilities available in the state of the art. This enables the consideration of reliability as an additional system optimization objective in a single and elegant way. To reduce the overall profiling overhead, a task execution time prediction mechanism utilizing machine learning and static code analysis is introduced.

- The exploration of the prediction of prospective system states, by among others analyzing past behavior, to enable proactively optimizing the system state. Hereby, past behavior is represented by information captured from the monitoring data. In this thesis particularly, two task prediction mechanisms, inspired by run-length Markov predictors, are developed. These mechanisms analyze past executions to detect task patterns and so, predict upcoming tasks.
- The study of methods to dynamically balance contradicting optimization goals based on both user inputs and alterations of the system state. Here, a hierarchical organic computing framework inlcuding an extended classifier system XCS and an offline rule generator is introduced. The framework is adapted to the needs of the mechanism employed in this thesis to influence the underlying system's behavior, task scheduling. In particular, a reward function based on a task cost simulator is provided.
- Investigating and developing new mechanisms to determine future system behavior by scheduling tasks ready-to-execute. This includes a mechanism to dynamically co-schedule multiple processes on the same computing node. The mechanism shares information via shared memory and pontentially reschedules waiting tasks after the arrival of new tasks in the system.

Ultimately, a core of functionalities and mechanisms, which realizes organic computing in a task-based runtime system and thereby enables a dynamic and proactive adaptation of the system, is studied and provided. This core is integrated and implemented in an existing runtime system and then evaluated extensively.

1.1 Thesis Organization

The structure of this thesis is organized as four parts. Part I includes the introduction and motivation of this thesis (s. Ch. 1) and gives a detailed problem statement (s. Ch. 2). Chapter 3 presents the necessary background and related work. Detailed explanations and definitions of proactivity, proactive adaptation, self-organization, and organic computing belong to the thesis' background. Furthermore, the runtime systems used in this thesis, HALadapt and EMB^2 are introduced. Chapter 4 finalizes Part I by elucidating the thesis' holistic approach and details its contributions to the state-of-art.

The theme of Part II is the system state, and mechanisms and methods that allow capturing and predicting system state descriptions. First, Chapter 5 presents some research projects that feature runtime systems for different heterogeneous systems and analyzes the projects' optimization goals, characteristics, requirements, and constraints for the runtime systems and heterogeneous platforms. Thereby, it sets the foundation for and motivates the remainder of this thesis. Chapter 6 introduces mechanisms to

monitor system behavior and, in particular, introduces a novel mechanism to heuristically measure system reliability. Finally, Chapter 7 introduces mechanisms to predict future system states, particularly upcoming tasks. So, this chapter builds the basis for the proactive behavior of the thesis' approach.

Part III focuses on methods to affect the future behavior of an underlying system. Chapter 8 thereby builds the heart of this thesis as the introduced learning mechanism utilizes the knowledge of the previous chapters to learn rules. These rules dynamically balance multiple optimization goals by associating weights to the optimization goals. The learning mechanism is based on principles of Organic Computing and utilizes a multi-level observer/controller framework. The resulting weights are then used to form an evaluation function for list task scheduling mechanisms. Task scheduling is discussed and new scheduling mechanisms for dynamic systems are introduced in Chapter 9.

The thesis concludes with Part IV that summarizes the thesis and discusses its contributions. Additionally, an outlook into future work and ideas for extensions to this thesis are given.

1.2 Collaborations

The results of three collaboration projects are part of this thesis. In particular, these are the following projects:
As part of the BMBF (Bundesministerium für Bildung und Forschung) funded Software Campus a project called Task-Scheduling für heterogene, parallele Systeme in Echtzeitumgebung (TahpSE) was carried out in cooperation with Siemens. The focus of the project was studying and integrating task scheduling algorithms for heterogeneous systems in the runtime system Embedded Multicore Building Blocks (EMB^2, see Sec. 3.4). Results were published [10, 11, 12].

A second BMBF funded project called "ENVELOPE – Effizienz und Zuverlässigkeit: Selbstorganisation in HPC Systemen" was conducted within the scope of the funding program "IKT 2020 - Forschung für Innovationen" in cooperation with JGU Mainz, RWTH Aachen, and TU Munich. The project goal was to provide proactive and lightweight mechanisms to improve system reliability in HPC systems while simultaneously providing efficiency in terms of energy and execution time and reducing complexity for application developers. Publications of several results exist [10, 13].

The third project focused on a runtime system for future automotive systems.

1.3 Previously Published Content

This thesis contains previously published papers:

Chapter 6: [14], [13]

Chapter 8: [15]

Chapter 9: [10], [16], [11], [12]

CHAPTER

TWO

PROBLEM STATEMENT

This chapter gives a detailed explanation and discusses the general problem that this thesis solves. In particular, it takes a closer look at historic developments and elaborates why the solutions presented in this thesis are needed.

Historically, new processing chips satisfied the ever growing need for more computational power mainly by increasing their operating frequency and the instruction-level parallelism (ILP). The main factor for this was constantly shrinking the design of a single Complementary Metal-Oxide-Semiconductor (CMOS) transistor. This trend was summarized by Gordon Moore into his famous statement, called Moore's Law [17], which says that the number of transistors on a single chip doubles every 12 months. In 1975 [18], Moore revised his original statement, increasing the duration of a doubling cycle to two years. Shrinking the transistor size simultaneously lead to a decrease in power consumption as stated by Dennard scaling [19]. This fact allowed processor designers to increase the operating frequency without increasing power consumption.

However, in the last 15 years the point was reached where Dennard scaling broke down. Increasingly smaller transistor sizes lead to rising power leakage, thereby generating additional heat. In combination with the Memory Wall [20], the widening gap in performance between processors and memory, and the instruction-level parallelism (ILP) Wall [21], the challenge to further extract parallelism form a sequential stream, the cost for increasing single thread performance was too high to stay economically viable.

Processor manufacturers then switched to increasing parallelism in the form of multi-core processors to satisfy performance needs. Instead of a single, complex processing core, multi-core processors consist of multiple copies of existing core designs. However, not all applications are able to benefit from constantly growing parallelism. This fact is best expressed by Amdahl's Law [22] (see Eq. (2.1)). Amdahl's Law gives a theoretical speed up limit for a single task t with execution time T, where $T = (1 - p) \cdot T + p \cdot T$ with p being the part of the task that would benefit from additional parallelism and $(1-p)$ being the sequential part of the task that would not benefit from parallelism. Hence, the

parallel execution time $T(n)$ of task t on n processing units is $T(n) = (1-p) \cdot T + p \cdot \frac{T}{n}$. This leads to the following speed up limit:

$$S(n) = \frac{T}{T(n)} = \frac{1}{(1-p) + \frac{p}{n}} \xrightarrow{n \to \infty} \frac{1}{1-p}, \tag{2.1}$$

where n is the number of computing resources used to parallelize the task.

Therefore, additional architecture design ideas were necessary to further satisfy the need for increasing performance. Such a new concept is heterogeneous computing, where different processing cores are integrated into a single system or chip. Hence, modern computer architectures include specific accelerators that are optimized for certain application characteristics. A prominent example are graphic processing units (GPUs) [23] that are used for computations traditionally performed by central processing units (CPUs) like matrix and vector operations. In general, GPUs with their massive number of simple processing cores and floating-point units are well suited for big and data-independent workloads that have a low amount of control flow instructions. The deployment of GPUs for general workloads is called general purpose computing on graphics processing units (GPGPU). Field-programmable gate arrays (FPGAs) [24] are another example that can be employed to accelerate specific algorithms, tasks or a set of instructions. FPGAs contain an array of programmable logic blocks that allow a designer to create configurations that can perform complex operations. Due to their parallel nature and energy-efficiency, FPGAs are optimal for certain applications like sliding-window applications.

However, heterogeneous architectures bring a whole new set of challenges with them. Using several different processors with varying architectural designs and instruction set architectures requires the usage of differing programming models by an application developer. NVIDIA introduced CUDA [25] as a programming language for their GPUs, which is based on the single-instruction-multiple-data (SIMD) concept. The SIMD concept is usually not deployed in the standard CPU programming languages. Similarly, FPGAs are also programmed differently compared to GPUs and CPUs. Hardware description languages like SystemVerilog [26] or Very High Speed Integrated Circuit Description Language (VHDL) [27] are employed to describe logic circuits. Hence, an application developer generally needs knowledge of several dissimilar programming models to utilize all processing units in a heterogeneous system. This, though, is not enough as a program has to be severely adapted to the underlying hardware architecture to be as efficient as possible. Efficient in the scope of this thesis can mean, e.g., minimum application execution time (makespan), minimum energy consumption, minimum heat dissipation, or maximum task throughput. So, deep platform knowledge is an additional necessity for an application developer.

A solution is offered by uniform programming models that provide a programming language for several different processing units. OpenCL [1] is an application programming interface (API) standard for this purpose. It provides programming languages, OpenCL C and Open CL C++, to write kernels that can then be dynamically compiled for different

processing devices. SYCL [4] is a higher-level programming model inspired by OpenCL and developed by the Khronos Group to simplify the programming of heterogeneous architectures. The uniform programming language is based on C++ and offers unified shared memory and a wide range of C++ features that can be used by the developer to create a heterogeneous application. Another example is Open Accelerators (OpenACC) [28], a programming standard for parallel computing on heterogeneous architectures. Similar to OpenMP [2], OpenACC provides pragmas to offload tasks to accelerators. Additionally, classic programming models OpenMP [2] and MPI [3] have been extended to include accelerator support in their latest versions.

However, writing applications for heterogeneous systems with a uniform programming model includes further complexity. Accelerators often posses dedicated memory, therefore an application developer has to manually handle data management, i.e., determining which data has to be transferred to which device at which point in time.

The aforementioned OpenCL and SYCL also solve this challenge by providing a unified virtual memory space, called Shared Virtual Memory (SVM) or Unified Shared Memory (USM). In OpenCL's 2.0 version, SVM allows the programmer to create buffers that can be both used by the host and other devices freeing the programmer from manually transferring data. USM in the newest SYCL 2020 version allows the programmer to naturally utilize pointers without buffers or accessors over multiple devices.

Nevertheless, none of these approaches is able to answer the question, when should a task or workload be loaded off to an accelerator instead of executed on the host CPU. For dynamic systems where the set of applications may change over time, and starting points and periods of tasks may be unknown, this question usually can not be answered at design time. Modern embedded systems allow the addition and removal of functionality in the form of apps, thereby regularly changing the set of applications to be executed. Similarly, users constantly send new jobs to HPC servers. In conclusion, new decisions have to be made dynamically. Furthermore, scheduling decisions are affected by the problem sizes of tasks and the current state of the system, e.g., the utilization and availability of processing units, which usually are also not known at design time and may change over the system's life cycle.

Runtime systems like HALadapt [5], StarPU [6], and the Tango framework [29] use scheduling algorithms to make scheduling decisions at runtime and therefore are able to include the aforementioned factors into their decisions. In general, these runtime systems are utilized to abstract an application from its implementations and the underlying hardware architecture. They implement a library-based approach that allows application developers to specify an abstract functionality, e.g., a matrix multiplication, and then provide several different implementation variations (kernels) for said functionality targeting different processing units and instruction sets. On the one hand this enables a modular application development process, where an application domain expert is allowed to focus on the application structure whereas hardware experts can optimize kernel implemen-

tations targeting different hardware. The runtime system, on the other hand, is able to select an optimized kernel concerning the aforementioned factors at runtime.

A solution that combines a unified programming model with a runtime system approach that is supported by a hardware concept is provided by the Heterogeneous System Architecture (HSA) [7]. HSA offers queues for the computing devices of a system and allows work stealing between the queues to balance load. Though, HSA has to be supported by the underlying hardware but as of yet only a limited amount of AMD or ARM architectures offer HSA support and features. Additionally, simple work stealing scheduling is not able to dynamically adapt to changing and new situations where different optimization goals become important. There also has been no further updates to the HSA specification, new press releases or publications since 2018.

The state of the art in runtime systems focuses mainly on optimizing makespan, and in the form of TANGO and StarPU also on minimizing energy consumption. Modern computing systems, however, are affected by a distinctly larger set of optimization goals and constraints. In addition, the set of optimization goals depends on the field of application in which a system is used (see Chap. 5 for an analysis of three research projects where heterogeneous architectures are deployed in different computing classes). HPC systems with their ever increasing complexity and number of components create the necessity to consider system reliability and availability as optimization goals because increasing the complexity and number of components leads to the rise of the failure rate of a single component. Heterogeneous architectures are integrated into modern embedded systems in the form of multiprocessor systems-on-chip (MPSoCs). Examples are the Xilinx Zynq Ultrascale+ [30], which comprises a quad-core ARM Cortex-A53 CPU, a dual-core ARM Cortex-R5 real-time processing unit, an ARM Mali-400 MP2 GPU, and a video codec unit, and the upcoming NVIDIA Parker [31] MPSoC, which consists of a NVIDIA Pascal GPU, 2 Denver 2.0 cores, 4 ARM Cortex A57 cores, and a video codec unit. Most embedded systems are subjected to the same constraints. They usually have to operate with limited resources. Typical limitations are the amount of available memory, which affects design decisions and the amount of memory that can, e.g., be used to create checkpoints to increase system reliability, and the amount of stored energy by accumulators, creating a necessity to consider energy consumption. Additionally, embedded systems are usually strictly limited in space like mobile phones or automotives. This leads to a restricted cooling capacity of the overall system, thereby generating a requirement to strictly reduce heat dissipation and in association energy consumption. A subset of embedded systems comes with safety constraints. Automotives and planes are examples where a failure may lead to catastrophic results. Therefore, specific tasks have to be executed within a set time span that has to be formally guaranteed by the system. Automotives in particular have to fulfill the Automated Safety Integrity Level (ASIL) standard, which includes four levels from ASIL A to ASIL D with increasing safety levels. Safety guarantees imply shielding the specific tasks and system parts from side effects.

The importance or weight of different, and often contradicting, optimization goals, though, is not set at design time and may change over the course of a system's lifespan. A big influence are environmental situations. For example a switched-off car is limited to its accumulator as an energy source, hence creating a strict energy budget. This has to be reflected by the optimization goals of the system. Environmental factors could also lead to the overheating of the system, generating a system-wide need to reduce load and possibly switch off resources. Ideally, such problems should never arise in a system. Especially embedded systems have to keep the user experience as flawless as possible and guarantee certain quality of service (QoS) standards during the system's whole life span. Avoiding faulty or disadvantageous states requires proactive mechanisms that avoid these system states before they occur. Proactive mechanisms have the potential to lessen the usage of emergency functionality that is called to prevent unsafe system states. A proactive mechanism could, e.g., delay the execution of uncritical tasks or lower their accuracy or QoS to avoid reaching a critical system temperature, if there is a high probability that executing all tasks ready-to-execute with the full computing power of the system may lead to a critical temperature threshold.

In summary, application developers need tools or frameworks that reduce the complexity of using heterogeneous architectures and enable an efficient usage concerning different and contradicting optimization goals while simultaneously being able to dynamically and proactively adapt the system, improve the system state, and avoid disadvantageous and faulty states.

The next chapter gives an overview over several approaches that have been used to tackle problems mentioned in this chapter. Chapter 4 then elucidates the approach and the contributions of this thesis in detail. Particularly, the relations and interactions between the different components implemented for this thesis to achieve proactive adaptation and self-organization are presented and explained in Section 4.2.

CHAPTER

THREE

BACKGROUND AND RELATED WORK

In this chapter, the necessary background for this dissertation and related approaches to hide the complexity of heterogeneous architectures and achieve adaptability and proactivity are presented and explained. Proactivity and proactive adaptation are defined and elucidated in Section 3.1. The second Section 3.2 introduces the concepts of self-organization and organic computing. In this thesis, these concepts are used as a basis to proactively adapt the underlying system. Section 3.3 and Section 3.4 describe the runtime systems used in this thesis. The mechanisms presented in this thesis are integrated into and evaluated with one of these runtime systems. Additionally, state-of-the-art solutions for hiding the complexity of heterogeneous architectures, and achieving adaptability and proactivity are discussed in Section 3.5.

3.1 Proactivity & Proactive Adaptation

The following section elucidates proactivity and proactive behavior by giving definitions and discussing existing work that implements proactivity in different fields of operation. In this thesis, system proactivity and proactive adaptation of the system are the overarching goals that shall be achieved by implementing concepts of Organic Computing. Organic Computing is discussed in Section 3.2.

Proactivity or **proactive behavior** is defined as acting before the relative need of the action arises [32]. So, proactivity is opposed to reactive behavior where an action is only taken after the causing event has occurred. The Merriam-Webster online dictionary[1] defines proactive as follows:
"*Acting in anticipation of future problems, needs, or changes.*" The definition of proactivity implies the necessity of a prediction mechanism, that is able to predict future events, in

[1] `https://www.merriam-webster.com/dictionary/proactive` (last visit 03/30/22)

order to implement proactive behavior. In the literature, several works have implemented proactivity in different contexts and fields of operation.

Klös et al. [33] present a profile-based approach to extend cyber-physical systems, in particular a hay fever medicine production system, with proactive adaptation using a MAPE-Knowledge feedback loop. Here, MAPE stands for monitor, analyze, plan, and evaluate. The approach monitors system and environment behavior and stores the parameter values in profiles. The profiles are then used to predict future behavior by comparing current observations with past profiles and selecting those that include the current values and an additional window of values that followed the observed values. If several profiles are selected, statistical metrics like the average, maximum, or a weighted average can be computed. The predictions and current observations are used to trigger specific rules that proactively adapt the system.

A Bayesian Network (BN) is used to model the relationship between context and adaptation costs in [34] in the context of a Observe, Orient, Decide, Act (OODA) model. Context is defined as "any information that can characterize the situation of an entity" by Abowd et al. [35], i.e., data and information that is usually provided by monitoring and observation. The BN provides probabilities for different cost functions considering the current context, which enables the calculation of cost risk functions. The cost risk functions are used in the decision making process that selects an appropriate action for the current context. The actual costs of the selected action is later monitored and provided to the BN as feedback. This approach was applied in a manufacturing scenario of the oil and gas industry.

In [36] proactivity in the context of program optimization is achieved by creating prediction models that correlate inputs with future program behavior. To create the models, Tian et al. use decision trees. This approach enables to dynamically optimize a program according to the prediction depending on the program input. For example, the approach is used to dynamically select an appropriate function version generated by the compiler at runtime.

Grosinger et al. [37] provide proactivity for agents by deploying an equilibrium maintenance algorithm based on fuzzy logic. This approach is utilized to make robots in a human-robot-interaction proactive. The algorithm forms a control loop that, first, estimates the current state and computes the equilibrium of this state. If the system is not in equilibrium, possible actions with favorable future benefit are computed. Future benefit is determined by states with fuzzy desirability. Out of the possible actions, one is selected and executed.

A framework for proactive fault tolerance was developed by Vallée et al. [38]. The framework consists of three components, a fault predictor, a policy daemon, and a fault tolerance daemon. The fault predictor analyzes system logs for disk errors and hardware sensor values for aberrant system temperatures. The policy daemon is activated when a fault is predicted and triggers the appropriate fault tolerance policy. The triggered policy

is then executed by the fault tolerance daemon. In the current state, virtual machine or process migration and pause/unpause is supported.

Kramer et al. [39] implement proactive, self-optimizing system behavior within an adaptive, heterogeneous architecture. Therefore, a system state is defined by clustering evaluated monitoring data. Hereby, an individual state is represented by a cluster. To predict future system states, a run-length encoding Markov predictor is deployed. Self-optimizing behavior is realized by learning optimization rules via a learning classifier system (LCS). The rules map a system state to an optimization action. Combining the prediction method with the LCS enables the system to proactively self-optimize.

In their work, Engel et al. [40] present a conceptual architecture for proactive event-driven computing. The architecture includes the two modules *predict* and *act*. Predict processes incoming event streams of current and past events and derives new and potentially future events. Act is initialized with an offline-generated world model that is adapted over time according to the event predictions. The world model is used to decide how to act based on the predicted events. Event processing agents are employed to implement the predict module. Engel et al. state three possibilities for a processing agent:

- Rule-based predictive agents: match a specific event pattern to derived events and potentially a time interval and occurrence probability.
- Bayesian agents: use a Bayesian network to predict future events.
- Classifying agents: use classifiers, e.g., decision trees or random forests, to derive events.

The act module is implemented by a proactive agent using a Markov decision process that gets derived events as input and based on this information decides which actions should be taken.

A framework for the support of proactive adaptation in pervasive systems is presented in VanSyckel's PhD thesis [41]. This thesis states three requirements that are necessary to support proactivity. First, the entity needs to be aware of its current context and possible future changes. Second, based on this information a decision about necessary adaptations that lead to a viable configuration instantiation has to be made. Third, these adaptations have to be enforced. The presented framework consists of a context management component that interacts with sensors and actuators and provides context prediction, an application configuration model, and an adaptation coordination component that coordinates the adaptations of multiple applications. Prediction is done by five different models, in particular alignment approach, linear regression, Markov model, self-organizing map, and state predictor. A model is chosen based on the parameters of the prediction task These parameters are data type, dimension, prediction horizon, quality of the data set, and quality of service. The application configuration model allows the

specification of applications' context dependencies, which enables the computation of adaptation alternatives, and a metric to rate application configurations, which is used to optimize adaptation decisions. The adaptation alternatives are searched via a depth-first search-based algorithm. All alternatives are then rated, whereby the best adaptations are determined and provided to the application.

In summary, the literature provides several concepts and methods to implement proactivity in different fields of operation. These can serve as inspiration for this thesis, especially regarding state prediction methods. However, the use case of this thesis, dynamically and proactively balancing contradicting optimization goals and adapting the system via task scheduling, is not solved explicitly in the state of the art. This thesis employs concepts of Organic Computing (s. Sec 3.2) to achieve proactivity in this context.

3.2 Self-Organization & Organic Computing

This section discusses self-organization and Organic Computing. The discussion gives definitions for both concepts and presents a general approach to implement Organic Computing with an observer/controller architecture. The concepts of self-organization and especially Organic Computing are used to achieve proactive adaptability of the underlying system. Section 4.2 later elaborates the thesis' holistic approach that uses Organic Computing concepts and is based on a more sophisticated observer/controller architecture.

Self-organization (SO) is defined by Camazine et al. [8] as follows:
"*SO is a process in which pattern at the global level of a system emerges solely from numerous interaction among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern*". In particular, there is no external ruling element that dictates the self-organizing process that creates a global order [42, 43, 44, 45, 46, 47].

Emergence is a concept tightly associated with SO. It is a characteristic of a whole system where the parts of the system form a pattern or behavior that cannot be reduced to the sum of its parts and is not existent on the element-level. There exists the famous quote by Aristotle "the whole is more than the sum of its parts". Emergence is therefore not only defined by the elements of the system, but by their interaction. In principal, emergence is a self-organized process, which leads from a state of lower to a state of higher order [9, 43, 47, 48]. Thereby, being self-organized is not a necessary characteristic in all available definitions [45].

However, order is not always a desired result [9]. Famous examples of undesired order are buildings getting destroyed by resonance vibrations [49]. Systems usually should have well-defined behavior according to a given specification and for the system to be predictable to at least a certain degree. Nevertheless, the necessity for a system to be

adaptable and dynamic still exists. The solution is so-called “controlled emergence” or “controlled self-organization”, where SO is applied to adapt a system to a specific requirement or goal [50, 48]. To achieve this, an incentive is needed that guides behavior towards this goal [43, 46]. As it is not predictable, which micro-level goals will eventually lead to which macro behavior, it is necessary to allow goal and objective modifications by an external authority if the system behavior is not acceptable. The system’s ability to correct its behavior itself against disturbances from outside or inside the system is called “robustness”. A system is robust if it can recover to (active) or remain in (passive) a state of acceptable behavior in the presence of disturbances [9].

Organic Computing (OC) is a concept to handle complexity in modern computing systems based on “live-like” properties [51, 9]. The term “Organic Computing” was created in the context of a workshop on future topics in computer engineering in 2002 by the special interest group “Computer Architecture” (ARCS - Architektur von Computer-Systemen) within the German Computer Science Society (Gesellschaft für Informatik, GI) [52]. OC systems apply self-* features like self-organization, self-adaptation, self-healing, self-configuration, self-optimization etc. to autonomously and dynamically adapt to current conditions of and changes to their perceived environment. Additionally, OC allows for explicit external interference in case of undesired system behavior in the form of goal adjustment [49, 9]. In total, the goal of OC is to guarantee the survival of the system in the presence of internal and external disturbances, basically accomplishing robustness.

To achieve these features, OC moves from a centralized system to a decentralized one, containing multiple interacting sub-systems. Interactions and control in dynamic systems require behavior monitoring and assessment. For this purpose OC provides a general observer/controller architecture [9, 49, 53].

3.2.1 Observer/Controller Architecture

OC first partitions the system into an internal control mechanism (CM) and the system part that is responsible for actually executing the intended task [9]. This part is called the productive logic of the system. The CM allows for external user intervention in the form of goal adjustment. Everything outside of these system boundaries is called environment and can be observed and influenced by sensors and actuators. The CM uses the environment observations and additional observations of the productive system (see the green lines in Fig. 3.1) to directly influence the productive system (red lines). This means that the system itself decides how to act in which situations. The user indirectly influences the behavior of the productive system by providing and adjusting goals. The CM then transforms these abstract user goals into concrete decisions. These decisions may theoretically be directly altered by the user (dotted red line). The system consisting

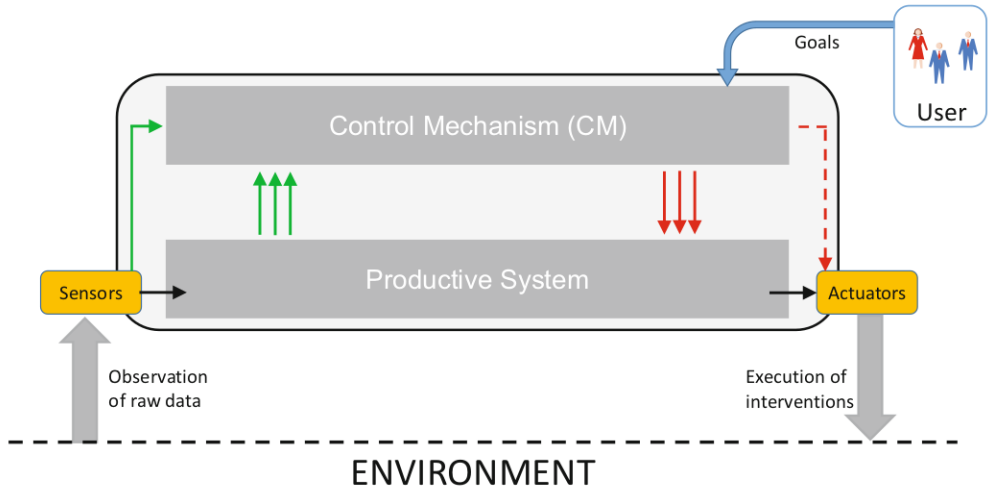


Figure 3.1: Abstract OC system architecture with a control mechanism and a productive system [9]

of the CM and the productive logic is exposed to unpredicted internal and external disturbances. This affects the system's capability to reach its goals. The CM has then to identify the best possible adaptation on its own.

By observing the environment and the system and determining the behavior of the productive system, the CM constitutes a control loop. As the CM also observes the resulting action of its decisions and their effects on performance, the control loop is closed. This general idea of a feedback-based control loop has been transferred to an observer/controller design pattern (see Fig. 3.2). The system is partitioned into three parts:

- The **System under Observation and Control (SuOC)** is the productive system part and remains functional even if the observer and controller are not working. It can consist of multiple independent entities or agents.
- The **observer** monitors the SuOC's and the environmental state based on a description provided by the controller, i.e., the controller decides what, in which frequency etc. should be monitored. This data is analyzed and then combined with, e.g., predictions into a description of the current situation. The situation is finally passed to the controller. Additionally, monitoring the environment provides a feedback of the actions selected by the observer.
- The **controller** evaluates the situation and based on this information selects actions to modify the SuOC in respect to the goals provided by the user. As the

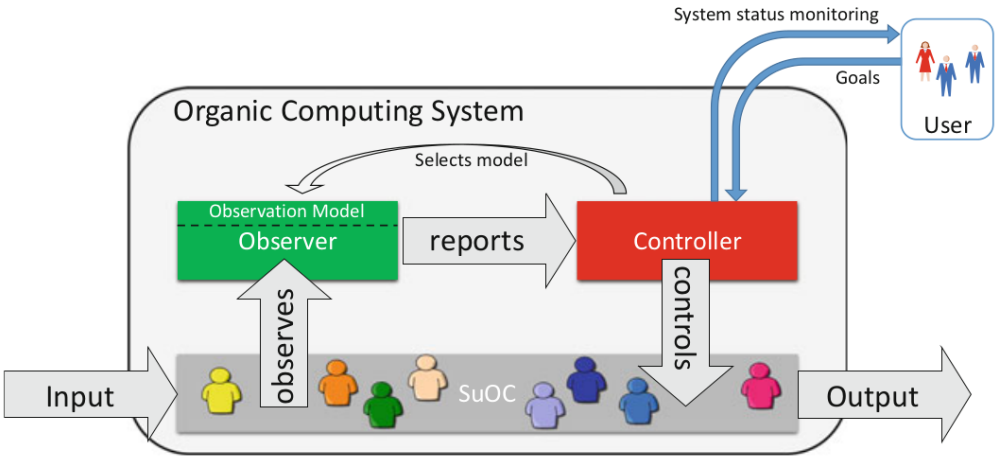


Figure 3.2: General OC observer/controller design pattern [9]

controller has to react to unknown and unexpected situations, its logic has to include trial-and-error concepts with techniques for generalization while simultaneously considering safety and quality of service constraints.

3.2.2 Relationship to Other Research Fields

Concepts very similar but with distinguished differences to OC have been frequently deployed in other research fields. This section presents some of the prominent relatives of OC.

In principal, the OC's observer/controller architecture can be mapped to a generic closed loop feedback control system [54] that is used in control engineering. In such a system, a reference value is compared to a current measurement, which is provided by a sensor, to compute the measurement's deviation from the intended value. This deviation is then used as input for a controller that computes new input for the controlled system. The resulting system behavior is then again monitored by the sensor. A common example for such a system is a PD controller. In OC, the observer is usually implemented by one or several sensors.

A prominent concept is Autonomic Computing [55, 56] and its MAPE loop where MAPE stands for monitor, analyze, plan and evaluate. This can be mapped to the observer/controller architecture of OC. The observer monitors and analyzes, and the controller plans and evaluates.

Another similar concept is the Sense-Plan-Act (SPA) paradigm in robotics [57]. SPA defines a simple control loop with three phases: gathering information from sensors

(sense), building a model and deciding the next move (plan), and executing the decided action (act).

In mechanical engineering the Operator-Controller Module (OCM) [58] is utilized to implement self-optimizing systems. OCM consists of three layers. The lowest level processes the productive system in hardware (similar to the SuOC). The middle layer contains a reflective operator that can adapt the operational layer based on given policies. This represents a closed control loop and can be mapped to the basic observer/controller loop of OC. Finally, the highest layer implements a cognitive operator that monitors the layer below and gathers information about itself and the environment. This information is used for learning mechanisms in order to self-improve the behavior employed by the middle layer.

Multi-Agent Systems (MAS) [59], consisting of several interacting, intelligent entities called agents, are used to heuristically solve complex problems by manifesting self-organization. Each agent has to be autonomous and only possesses a local view. Additionally, no central control entity may exist. Compared to OC, MAS is more of a meta-term that covers a variety of computational or socio-technical systems, but is not concerned with the transfer of design-time decisions to runtime, which is a major focus of OC [9].

3.3 HALadapt

This section elucidates the runtime system mainly used in this thesis. It was developed at Karlsruhe Institute of Technology at the Chair of Computer Architecture and Parallel Processing by Mario Kicherer and is called HALadapt.

HALadapt [60, 5, 61] is a task-based runtime system that operates on the user-level. It utilizes a library-based approach to separate an application from its implementation and abstract the underlying hardware architecture. An overview of this functionality can be found in Fig. 3.3. The user simply defines his or her kernels representing a specific functionality, e.g., a task like a matrix multiplication, and provides either one or multiple implementation variants targeting different processing units and programming models. HALadapt does not impose a particular limit in terms of granularity for these kernels, leaving it to user preference.

Supporting several processing units and programming models increases the dependencies an application has to fulfill in order to be executable. As this limits portability, HALadapt uses a decoupled development concept that outsources implementations into a specific repository consisting of separate libraries. This way, the application is freed of its dependencies and only the repository libraries depend on hardware-specific libraries. At runtime, HALadapt is then able to only load libraries that satisfy their dependencies guaranteeing the executability of the application.

To still ensure independence of the underlying hardware architecture, the runtime system offers a plugin API that abstracts the communication with processing units. The API

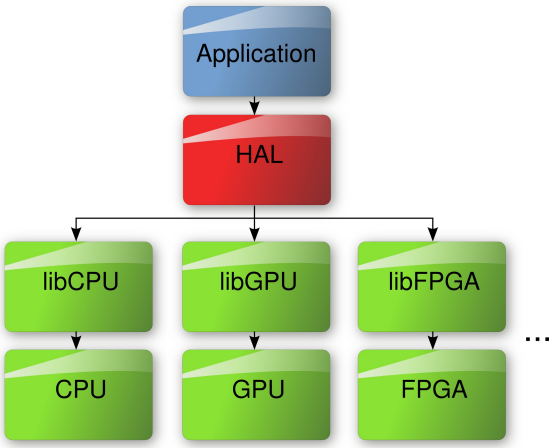


Figure 3.3: Overview of HALadapt [62, 61]

includes functionality to, e.g., query the number and type of available devices or execute data transfers (memory management is explained in Sec. 3.3.2 in detail). Again, these plugins are only loaded if their dependencies are satisfied. An example architecture is shown in Fig. 3.4.

At runtime, HALadapt then is able to select one of the available implementation variants for execution. This decision process can be made for a single task or a set of tasks combined into a task graph. As sophisticated decisions need information about the tasks to be executed, HALadapt uses a profiling mechanism that monitors past execution behavior (see Sec. 3.3.1). The profiling mechanism uses a feature that allows the execution of additional functionality, e.g., measuring the execution time of a task or computing a task schedule. This feature is implemented as a so called *call stack* which allows the dynamic registration of additional functionality to be executed before or after the original kernel function. A function added to the call stack is called *call stack entity* (CSE). Fig. 3.5 shows an example for the execution of a single task with HALadapt. First, the application transfers control to the runtime system by calling an entry function. HALadapt then executes all functionality added to the call stack. In this example, CSE A executes code before and after the target kernel, and CSE B only executes code before the actual execution. A special CSE, the target call CSE usually added last to the call stack, executes the actual kernel.

The implementation selection is conducted by task scheduling algorithms. HALadapt contains several scheduling algorithms, including the Heterogeneous Earliest Finish Time (HEFT) algorithm and a Simulated Annealing (SA) scheduling algorithm. Additionally, a developer can add its own scheduling algorithm as a plugin similarly to the hardware plugin mechanism mentioned before. Simple algorithms like HEFT schedule

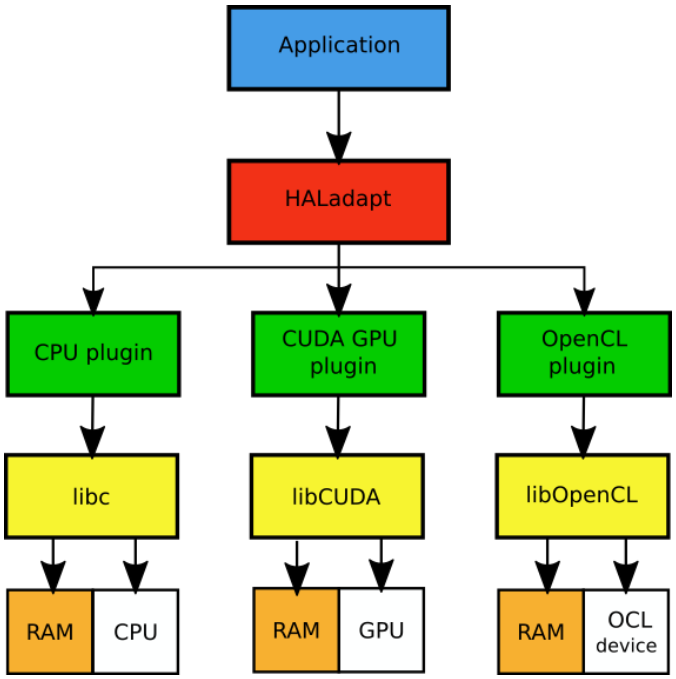


Figure 3.4: Plugin interface for different hardware devices [60]

tasks one after the other, which allows the runtime system to gradually alter the global system state without the need to reverse it. However, complex algorithms evaluate multiple task schedules and alter already made mapping decisions. To support such algorithms, a container concept that allows the online simulation of several task schedules has been integrated into HALadapt. This so-called container concept is explained in detail in Sec. 3.3.3.

HALadapt additionally offers cost awareness over multiple runtime system instances. For every processing unit in the system, a waiting queue is added to shared memory where different instances are able to insert their tasks. So, an instance wanting to insert new tasks is able to gather the availability of the processing units and include it in its scheduling decisions.

3.3.1 Cost Awareness Based on Past Behavior

Sophisticated implementation selection decisions require knowledge about the potential costs and benefits of the specific variants. To gain this knowledge, HALadapt uses online training [62]. Each kernel execution is measured with a plugin-based sensor interface. Different sensors, e.g., monitoring specific hardware, can be registered similar to the hardware plugins mentioned above. Included in HALadapt is a sensor that measures ex-

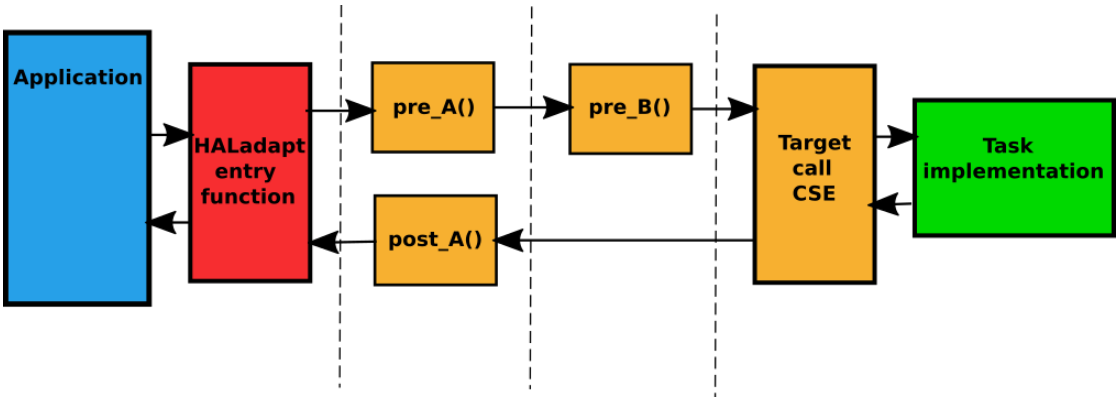


Figure 3.5: Example of a call stack for a single task [60]

ecution time by feeding the current wall clock time to the runtime system. Another sensor uses Intel's Running Average Power Limit (RAPL) interface to provide an estimation of the current energy and power consumption of modern Intel CPUs and GPUs.

The resulting measurements are stored in so-called cost vectors. For every sensor available on the executing processing unit a vector is created. A vector is associated with the problem size that has to be provided by the developer. The problem size can be, e.g., the dimensions of a matrix or the number of non-zero elements for a matrix multiplication. Future use of these measurements is enabled by storing the vectors in a database that combines all vectors of a processing unit group implementation pair. To reduce the need for actual execution, HALadapt uses inter- and extrapolation between known problem sizes to estimate costs for unknown problem sizes.

3.3.2 Memory Management

Accelerators in heterogeneous systems usually posses their own dedicated memory separate from the main memory. Therefore, data transfers between main and device memory are necessary to make the data needed for execution available on devices. As data transfers create additional overhead, this overhead has to be considered when implementation variants are selected. For a single kernel execution on a device, the data has to be transferred to the device before the execution and back to the main memory afterwards. However, if multiple kernels are executed in sequence on a device, this scheme results in unnecessary transfers and thereby overhead.

To solve this problem, HALadapt allows the developer to register the application data via a specific API call. For each data block, the runtime system allocates a management structure that contains information like the start address or the size of the data in host RAM. Additionally, a version counter is associated with each data block. The counter is

increased by write accesses, thereby enabling HALadapt to find the most recent copy of specific data. Thus, the runtime system is aware of the data and can track its location and possibly existing copies in the system. So, it is possible for HALadapt to detect when a data transfer is required and HALadapt then can initiate the transfer.

3.3.3 Online Simulation of Task Schedules

Simulating several task schedules and altering already made decisions requires either constantly altering the global state, which can lead to complex operations and severe overhead, or working with a copy. HALadapt chose working with copies by using a so-called container concept. Containers are organized hierarchically with a root container, that represents the global state, at the top. An example can be seen in Figure 3.6. Every

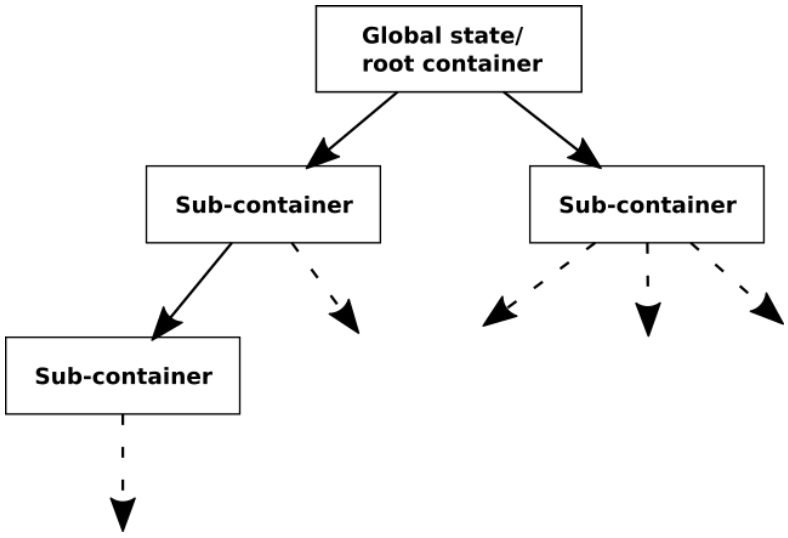


Figure 3.6: Example of a container hierarchy [60]

container can have an arbitrary number of sub-containers. A sub-container may contain new, removed, or modified objects of its parent. Each container for itself represents a copy of the global state, which means every alteration made in the context of this container affects the global state but is only visible for this container and its sub-containers. So, it is possible that a scheduling algorithm modifies several sub-containers simultaneously and afterwards compares the resulting system states. The best result can then be merged into the parent container, making the changes visible globally.

3.4 Embedded Multicore Building Blocks (EMB^2)

Here, the runtime system Embedded Multicore Building Blocks (EMB^2) is shortly presented. The dynamic scheduling algorithms presented in Section 9.2 are integrated into EMB^2.

EMB^2 [63] is a C/C++ library and runtime system for parallel programming of embedded systems.[2] One of the challenges EMB^2 aims to solve is to abstract from the complexity of the underlying hardware by hiding platform-specific details and making the development of applications portable. For this, EMB^2 provides fundamental functionality for creating and synchronizing threads and memory management. These functions are implemented as wrappers that wrap features specific to the underlying system. The runtime system also provides atomic operations that are directly mapped to the instruction set of the target processor.

Additionally, EMB^2 builds on MTAPI [64], which like HALadapt defines a task model that allows several implementation variants for a given task. A user just defines a specific functionality, e.g., a matrix multiplication, and provides one or multiple implementations for this task targeting different processing units and programming models. MTAPI allows a developer to start tasks and to synchronize on their completion, where the actual execution is controlled by the runtime system.

To be able to benefit of possible heterogeneous implementation variants, EMB^2 offers the task management and scheduling support defined in MTAPI. The runtime scheduling implementation of EMB^2 distributes the task instances between heterogeneous processing units based on the number of already scheduled instances of the same task. For homogeneous multicore CPUs, an additional work stealing scheduler [65, 66] is used to balance the load.

Furthermore, EMB^2 offers building blocks for typical parallelization tasks like loops and templates for stream-based applications. These algorithm and dataflow building blocks create tasks that are then guided to and executed by the MTAPI task scheduler. EMB^2 also provides lock-/wait-free concurrent containers to guarantee progress for real-time systems. The high-level architecture of EMB^2 can be seen in Figure 3.7.

3.5 Related Work

In the literature, a variety of programming models that support heterogeneous architectures exists. The most well-known are OpenMP [2], OpenCL [1] and MPI [3]. However, none of these supports the programmer in his decision where to execute a task.

The research community also developed new programming models for heterogeneous architectures. OmpSs [67, 68] and OmpSs-2 [69] are programming models based

[2] `https://embb.io/` (last visit 03/30/22)

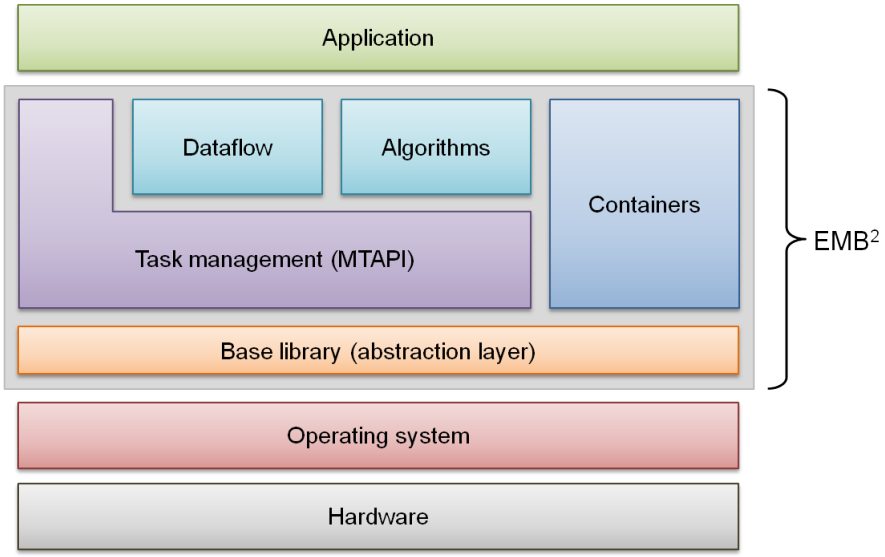


Figure 3.7: High-level architecture of EMB^2 [63]

on the source-to-source compiler Mercurium and the runtime libraries Nanos++ [70] and Nanos6 [71], respectively. They combine OpenMP and StarSs [72], whereby the execution model of StarSs, a thread-pool rather than OpenMP's fork-join parallelism, is used. OmpSs offers pragmas to parallelize sequential code and to define and offload tasks to accelerators. Again, the application developer is allowed to provide several implementation variants for a defined task. The runtime system is then able to select the task version that minimizes its finish time for known data sizes [73]. Additionally, OmpSs uses a different memory model compared to OpenMP. It assumes that multiple address spaces may exist and therefore shared data may reside in locations that are only accessible by a subset of the devices in the system. OmpSs provides a possibility to explicitly mark shared data and then manages necessary data transfers and coherency. Compared to the approach of this dissertation, OmpSs only focuses on execution time.

TANGO (transparent heterogeneous architecture deployment for energy gain in operation) [29, 74] is a research project focused on providing tools and technologies to facilitate the adoption of new heterogeneous hardware. Djemame et al. propose a system architecture (see Fig. 3.8) consisting of three layers spanning from remote to local processing capabilities. The goal of layer one is to facilitate the modeling, design and construction of applications and is partitioned into the following three components:

- Requirements and Design Modeling: Helps the developer to better understand deployment alternatives in particular situations targeting Quality of Service, Quality of Protection, cost of operation and power consumption behavior. This is done by rapid prototyping on actual hardware or emulators.

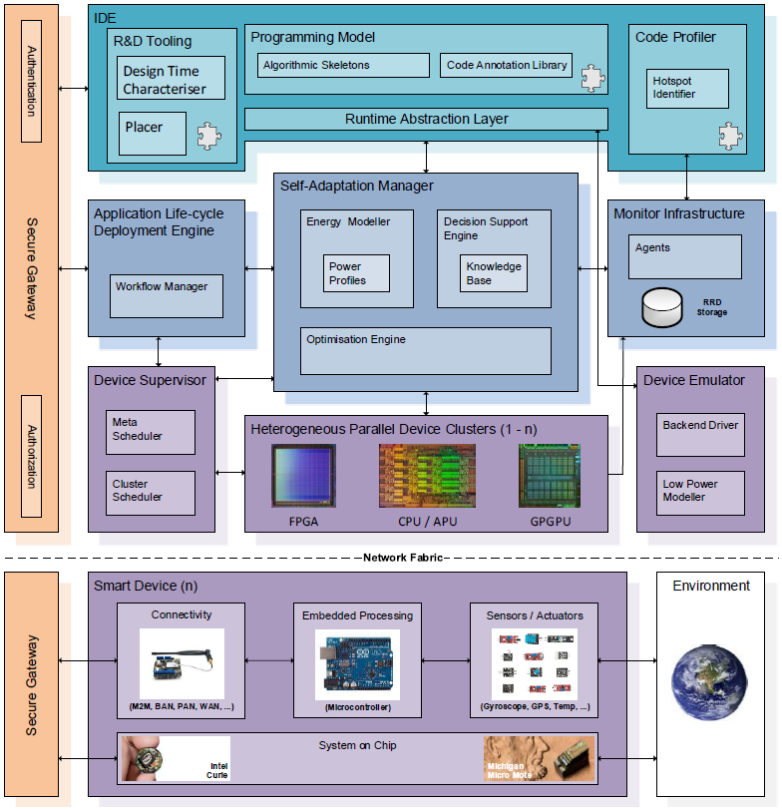


Figure 3.8: System architecture of TANGO [75]

- Programming model (PM): The PM used is a combination of the task-based models OmpSs and COMPSs [76]. COMPSs is a framework for parallelizing applications written in Java, C/C++ and Python and distributing tasks to nodes in a cluster, clouds, or grids. The combination allows for coarser and finer grain tasks and different implementation versions for a specific task. The programming model is complemented with a runtime for distributing coarse- and scheduling fine-grain tasks.

- Code optimizer: Helps the developer understand the energy foot print of the written Java code by static analysis and profiling.

The second layer provides components that handle the placement of applications considering energy models on target heterogeneous parallel architectures:

- Application Lifecycle Deployment Engine (ALDE): Uses containers to simplify all HPC development steps from compilation, packetization, deployment to actually running applications. After compilation and packetization, the user can upload

the application to any testbed managed by ALDE and define different execution configurations similar to Cloud computing. In combination with the Self-Adaptation Manager, the execution can be adapted and made more efficient.

- Self-Adaptation Manager: The Self-Adaptation Manager [77] sits at the heart of TANGO's runtime framework. It is event-driven collecting information from several other components and decides for each event what adaptation to apply. Events such as an idle host, a host failure or boundary conditions trigger responses such as starting/stopping applications, redeploying applications or adapting wall time.

- Monitor Infrastructure: Monitors the devices and provides current and past execution status metrics for applications.

The last layer provides an extension to SLURM, an open-source cluster resource-management and job scheduling system, called Device Supervisor, and a Device Emulator, an implementation of a new scheduling method that combines simulation to generate execution costs with the actual scheduling process. Compared to this work, TANGO mainly focuses on energy and execution time. The adaptation approach of TANGO also differs from this thesis by directly redeploying applications instead of creating a new balancing of multiple optimization goals and computing a new schedule based on the new weights.

A well-known task-based runtime system for heterogeneous multicore architectures is StarPU [6]. StarPU abstracts the underlying hardware by offering offloadable tasks called *codelets*. For a codelet, the programmer can provide several different implementation versions. At runtime, StarPU selects the best performing version for each input size. The decision relies on performance models that are created dynamically based on previous executions [78]. The performance models are associated with a hash value computed with the layout and size of the data of the executed kernel to distinguish between different kernel instances. Furthermore, StarPU manages necessary data transfers with a directory-based MSI protocol [79]. Additionally, filters are used to hierarchically partition data as devices usually only work on a subset of the input data. In contrast to this work, StarPU only considers energy consumption as an additional optimization goal. Furthermore, the energy consumption can only be optimized if the application developer provides an energy model to StarPU and the balancing of the optimization goals is not dynamically adapted to new situations.

HPX [80] is a runtime system for parallel and distributed applications and implements the execution model ParallelX. HPX consists of five components: a thread manager, an active global address space (AGAS), parcel, and global and local performance counters used for monitoring. The thread manager schedules light-weight user-level threads on the CPU and is able to choose from several algorithms including work stealing. AGAS hides explicit message passing and parcel is a mechanism to support active messages,

i.e., remote thread invocation. HPX also supports offloading computations to CUDA GPUs [81]. However, offloading has to be done explicitly by the programmer.

Legion [82] is a runtime system based on a data-centric programming model. It employs tasks as an abstraction of a unit of parallel execution and logical regions to support a relational model for data. The data regions posses an index space and fields. These are referred to as rows and columns. The regions can either be partitioned based on their index space or spliced on their field space. A legion program executes a tree of tasks that are created recursively, thereby creating parallelism. Each task works on a specific logical region. Legion also allows offloading, but the decision where to execute a task has to be made by the programmer.

CHAPTER

FOUR

AN APROACH FOR PROACTIVE ADAPTATION IN SELF-ORGANIZING TASK-BASED RUNTIME SYSTEMS

This chapter presents the general approach and contributions of this thesis that close a gap in the current state of the art in the literature. It states the individual goals that are achieved within this work.

4.1 Contributions

The main goal of this thesis is to study OC mechanisms that enable a dynamic and proactive system adaptation and to integrate them into an existing runtime system for heterogeneous systems. Thereby, a runtime system is created that adapts to new and unexpected situations while simultaneously reducing system complexity and maintaining efficiency concerning multiple optimization goals like energy, temperature, and system reliability. To achieve this, the following research objectives and respective solutions to these problems are included in this thesis:

I. Analysis of task-based runtime systems in different use cases and computing classes of heterogeneous systems.

   Here, the goal is to find and identify the specific optimization goals, constraints, and requirements of the individual use cases and computing classes. Hence, conclusions can be drawn about restrictions, configurations, parameters, and objectives of future systems to be created. Additionally, this analysis motivates the necessity of this thesis by providing examples of dynamic systems with multiple optimization goals that require proactive adaptation at runtime.

II. A thorough study of methods and tools to capture the system state by monitoring the execution behavior and the system environment, and the introduction of new possibilities to evaluate the system and reduce profiling overhead.

    A meaningful and sufficiently descriptive system state capturing may require much data and information. So, a goal of this research point is to find ways to reduce the overhead created by collecting this data, in order to alter the execution behavior as little as possible. To extend the profiling possibilities available in the state of the art, a heuristic reliability metric based on the light-weight concept symptom-based fault detection is created. Thereby, this thesis is able to consider reliability as an additional system optimization objective in a simple and elegant way. A method to reduce the overall profiling overhead is introduced in the form of an execution time prediction mechanism based on machine learning and static code analysis.

III. The exploration of ways to augment the system state description by the prediction of prospective system sates.

    System state prediction allows the system to act proactively and take measures to avoid disadvantageous states before they manifest themselves. In this thesis, the prediction is done by the analysis of past behavior. Hereby, past behavior is represented by information gained from the monitoring data. Particularly, this thesis provides two prediction mechanisms, inspired by run-length Markov predictors, that predict upcoming tasks. The mechanisms scan past task executions for patterns and on this basis predict upcoming tasks. Additionally, the profiling database is deployed to inter- and extrapolate costs for unknown thread numbers and problem sizes of tasks already profiled.

IV. The research of methods to dynamically balance contradicting optimization goals based on both user inputs and alterations of the system state.

    A hierarchical OC framework that includes an extended classifier system XCS is utilized to learn rules that map a system state description to a weighting of the system's optimization objectives. The resulting weights for the different optimization goals are then used to form an evaluation function. This evaluation function can be deployed to, e.g., evaluate task schedules or other mechanisms that influence system behavior.

V. Studying ways to influence future system behavior via task scheduling.

    Here, different scheduling scenarios are looked at. Tasks belonging to one single process and tasks belonging to multiple different processes in dynamic systems are the focus of this thesis. For the single process scenario, a mechanism

that dynamically adapts task priorities according to the current system utilization to avoid starvation and improve the overall makespan is introduced. To coordinate multiple processes, a co-scheduling mechanism employing communication via shared memory is created. This enables the processes to share the necessary information to re-schedule and improve scheduling decisions after the arrival of new processes and tasks.

These research objectives are all steps that combine into an OC system able to dynamically adapt and optimize its behavior. The following section elucidates how these parts are combined and interact with each other to create such a system.

4.2 Bringing It All Together - The Holistic Approach

The holistic approach of this thesis is displayed in Figure 4.1. The approach is based on the Multi-Level Observer/Controller framework by Müller-Schloer et al. [9]. It consists of a monitoring and data analysis module, a controller and offline rule generation module, a task scheduling module that is able to simulate task schedules as a reward computation, and the rest of the system that is observed and influenced by the aforementioned modules.

The monitoring and data analysis module (s. the green components in Fig. 4.1) is built with three individual components. A monitoring component (called "Monitor" in Fig. 4.1) observes the underlying system and the environment of the system to create a snapshot of the current system state. Such a snapshot can include various execution metrics, e.g., processing unit utilizations, system temperatures, energy consumptions, task execution times, and a description of the environmental situation. An environmental situation can, e.g., be the operating phase of the system. If the engine of an automobile is turned off, its energy capacity is strictly limited by its battery. A similar environmental factor is the current loading capacity of the accumulator of a portable system. The monitoring information that can be associated with an executed task and the belonging processing units is stored in a database and associated with the specific task and its execution parameters. Execution parameters are, e.g., the problem size of the input parameters or the number of threads used for execution. The details of the monitoring component and the database are outlined in Chapter 6.

The stored information can later be utilized by the analysis and prediction component (called "Data Analysis & Prediction" in Fig. 4.1) to predict possible future states, and to augment the current system state. Predicting future system states can be done by the analysis and prediction component by, e.g., detecting patterns in past behavior. These predictions are added to the system state snapshot allowing the controller to proactively react to future events. In particular, two mechanisms that predict upcoming tasks and additional methods to predict task costs are developed. The prediction of system states

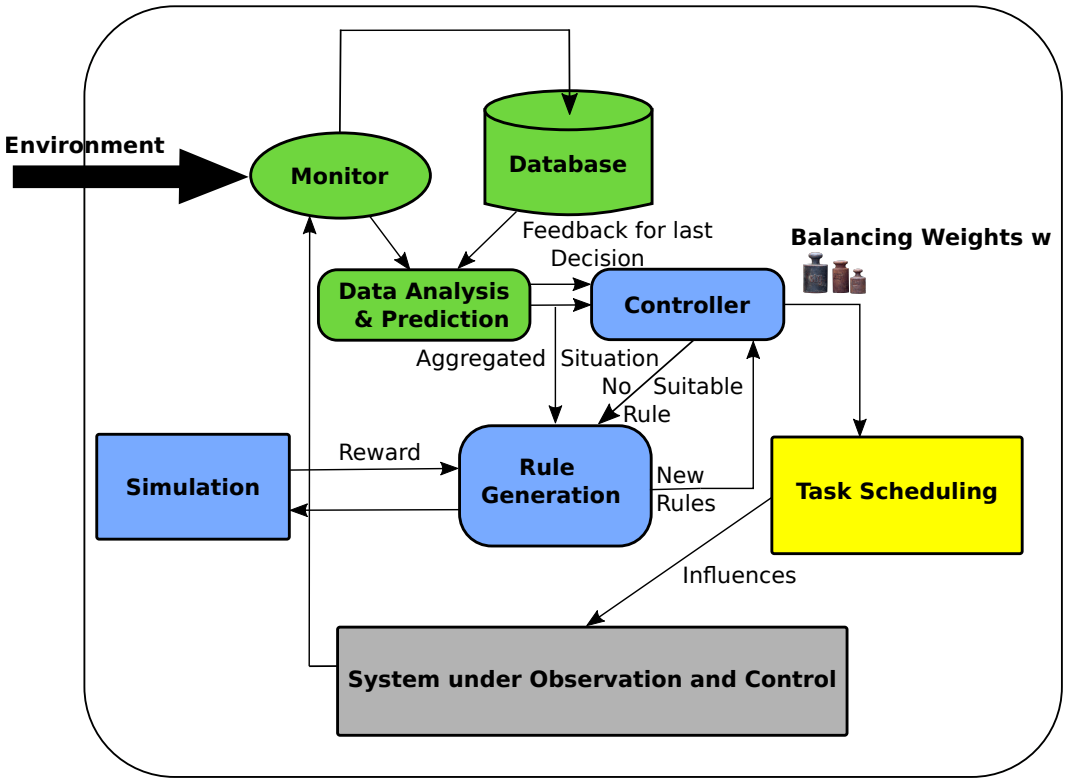


Figure 4.1: View of the holistic approach of this thesis

is elucidated in Chapter 7. The analysis component is also deployed to create feedback, i.e., a reward signal, for the previous controller decision. This is done by evaluating the current system state snapshot that has been affected by the decision of the controller and then providing this evaluation to the controller. The reward computation in combination with the controller is discussed in Chapter 8.

The task of the controller is to generate a balancing of the multiple contradicting optimization goals of the system out of the provided and augmented system state snapshot. Thus, the controller associates a weight with each optimization goal, which allows to compare improvements and deteriorations in one optimization goal to improvements and deteriorations in others when an action affecting the system is evaluated, e.g., a task schedule. To match a system state to a weighting, a set of rules is used. If no rule is available for a specific system state snapshot, the offline rule generation is triggered. The rule generation component utilizes the ability of the runtime system HALadapt to simulate the execution of task schedules and its effects on the system (see Sec. 3.3.3) to evaluate the new candidates. Therefore, it is possible to create new rules for the controller without executing rule candidates in the real system and risking possible safety and system constraint violations. The specifics of rule generation are explained in detail in Chapter 8. The controller and the offline rule generation are depicted as blue components in Fig. 4.1. Fig. 4.1 represents HALadapt's task simulator with the orange component "Simulation".

The optimization goal weights provided by the controller are finally used to create an evaluation function for a task scheduling module (the yellow component in Fig 4.1) as task scheduling is an effective way to dynamically influence and optimize the efficiency of a computing system. This thesis focuses on two different dynamic scheduling scenarios. In the first scenario, all tasks belong to a single process which means a single runtime system instance possesses all information about which tasks are about to start. For this scenario, this thesis provides a mechanism that improves task scheduling in the presence of task priorities. In order to avoid starvation and improve the overall makespan, this mechanism dynamically adapts the priorities of task instances according to the current system load. The second scenario considers the distribution of tasks over multiple processes that are not started simultaneously. This means that there is no single runtime system instance that possesses all information. Hence, additional communication between the multiple instances is required. Details can be found in Chapter 9.

Part II

The System State

CHAPTER

FIVE

REQUIREMENTS, CONSTRAINTS & OPTIMIZATION GOALS

This chapter presents three different research projects, in which a task-based runtime system is deployed in a specific computing class using heterogeneous architectures for their systems. The focus of the chapter is the analysis of the different computing classes and their special environments. The chapter discusses the diverse constraints, requirements, and optimization goals particular to these computing classes that originated out of this analysis. In doing so, this chapter provides motivation and needed information for the remainder of this thesis. In particular, it provides information about which optimization goals have to be considered and therefore which data should be collected by the profiling mechanism presented in Chapter 6 to describe the system state.

The first section 5.1 reviews the usage of HALadapt (see Sec. 3.2) in the context of HPC systems. Further, the deployment of EMB^2 (see Sec. 3.3) in non-safety critical embedded systems is discussed, and the section presents a project where a task-based runtime system is developed for automotive systems. Section 5.2 summarizes the insights gained in this chapter and outlines their impact for the remaining thesis.

5.1 Task-based Runtime Systems in Different Heterogeneous Systems

In a collaboration project called Envelope [1] funded by the Bundesministerium für Bildung und Forschung (BMBF), the runtime system HALadapt is used in heterogeneous HPC systems. The project consortium includes JGU Mainz, RWTH Aachen, and TU Munich as partners.

[1] envelope.itec.kit.edu

Modern HPC systems satisfy the ever growing need for an increase in performance with a constantly rising number of computing nodes. In combination with the steadily progressing miniaturization of components, this leads to an increasing failure rate of a single component. Classical methods that provide fault tolerance and increase system reliability, like checkpoints in a parallel file system and redundancy, are not enough to efficiently execute highly parallel applications [83]. Redundant components increase acquisition and operation costs, whereas checkpoints in a parallel file system raise the demands on the global memory system. This contradicts the goal to maximize system efficiency. For modern HPC systems it is therefore necessary to be able to proactively detect component failures and to use lightweight failure methods that increase system reliability.

As the name suggests, a main focus of HPC systems is maximizing performance. In this context, performance usually means maximizing job throughput or minimizing application makespan. This can be achieved by globally optimizing where and when jobs are executed.

The development of HPC systems is limited by its energy consumption and the correlated available cooling capacity and operational costs [84]. In the age of climate change, minimizing energy consumption is a major concern due to environmental reasons [85]. Energy consumption always leads to emissions thereby creating a global-warming greenhouse footprint. Additionally, an increase in energy consumption leads to more waste heat. This in turn then requires greater cooling capacity, which is limited by space and operational budget. The operational budget is additionally consumed by energy costs as well. In summary, minimizing energy consumption and maximizing energy efficiency is a major concern for HPC systems.

In a cooperation with Siemens, we used heterogeneous architectures and the runtime system EMB^2 [86] in the context of non-safety-critical embedded systems with soft real-time constraints [10]. A critical constraint for these systems is given by their dynamic nature. Tasks can be triggered by recursion, signals, or user interactions and therefore, not all task information is readily available. Hence, the runtime system and especially its scheduling mechanism have to be dynamic and adaptable. Another focus for Siemens is to implement soft real-time capability within EMB^2. This is achieved by allowing the developer to associate priorities with their kernels, which have to be considered during task scheduling.

Due to space restrictions in embedded systems, available resources in such systems are often limited. Typically, memory is strictly limited compared to HPC or desktop systems, requiring awareness of the total memory consumption. Therefore, dynamic memory allocation should be avoided. Limited resources are also a major factor considering energy consumption and heat dissipation. Embedded systems usually cannot provide the cooling capacities of HPC or desktop systems, thus restricting the available computing capacity and the amount of energy that can be consumned. Additionally, mobile

systems often use accumulators that can only provide a certain amount of energy. This creates the need to maximize system efficiency.

The third project uses heterogeneous architectures and a task-based runtime system in the field of automotive systems. For a subset of the functionality, safety constraints in the form of Automotive Safety Integrity Levels (ASIL) defined by the ISO 26262-1 [2] have to be guaranteed. ASIL contains four levels with ASIL A dictating the lowest and ASIL D the highest safety requirements. Safety requirements also imply guaranteeing strict deadlines for specific tasks. This means that certain functionality has to be shielded from side effects potentially caused by, e.g., operating systems or other tasks.

Automotive systems belonging to the field of embedded systems also suffer from space limitations and thereby face similar constraints for energy consumption and heat dissipation as mentioned above. Additionally, future automotive systems are not static and new situations and application updates can constantly occur. Therefore, dynamics and system adaptability are significant aspects.

5.2 Summary and Conclusion

This chapter looked at three research projects that each use a runtime system for heterogeneous architectures in a specific computing class. The focus of the projects are HPC systems, non-safety-critical embedded systems, and automotive systems, respectively. Each of these computing classes is defined by particular characteristics. The purpose of this chapter was the analysis of the optimization goals, requirements, and constraints that are dictated by the deployment in these computing classes. In summary, system efficiency concerning energy, heat dissipation, and performance is an important optimization goal in all computing classes. Additionally, dynamics is an important factor as new situations can occur at indeterminate points of time. However, there are also constraints and optimization goals unique to specific computing classes. Safety constraints are an example of a constraint that is of no concern in HPC and non-safety-critical embedded systems, but play a major role in automotive systems. So, runtime systems have to be adaptable to be useable in different computing classes.

Generally, the detected optimization goals and constraints contradict each other. Improving one goal or implementing a constraint usually leads to the deterioration of other optimization goals. Therefore, a balance between the different goals has to be found. The importance of optimization goals, however, depends on the current situation. Thus, it is not possible to statically set a balance for the total system runtime. In conclusion, optimization goals and constraints need to be dynamically balanced to optimize the system in different situations.

[2] https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en

CHAPTER

SIX

CAPTURING THE SYSTEM STATE

This chapter focuses on the components of my approach that are used to capture a snapshot of the current system state. A snapshot may consist of several execution and runtime metrics like the execution time, utilization, processing unit availability, energy consumption, or system and component temperatures, and metrics describing the system environment, e.g., GPS coordinates, or the room temperature. The actual metric selection, hereby, is not fixed and is dependent on the field of application and use case.

These measurements are stored inside a database and so can be used later for analysis and to augment future snapshots. Additionally, monitoring the system state and behavior enables the evaluation of actions taken to influence the system. The database can also be utilized to predict the costs of upcoming tasks, either as measurements are already stored in the database or by inter- and extrapolating known measurements. In the holistic approach presented in Section 4.2 this chapter represents the green marked components Monitor, Database, and Data Analysis & Prediction. The Prediction component is further extended in Chapter 7.

A short motivation for system behavior capturing in the context of this thesis and a presentation of related work is given in Section 6.1 of this chapter. The focus of the second section, Sec. 6.2, are the different methods employed to capture the metrics. Hereby, the extensions to the monitoring component of HALadapt (see Sec. 3.3.1 for the detailed introduction of HALadapt's monitoring component) that are made in the scope of this thesis are presented.

As there is no single hardware or performance counter available that represents the reliability of a system or its components in today's computing systems, Section 6.3 introduces an online heuristic metric to measure reliability created for this thesis. The heuristic metric thereby deploys the concept of symptom-based fault detection to compute an online fault rate.

The overhead of profiling mechanisms plays an important role in their evaluation of usability in a system. Therefore, the fourth section 6.4 elucidates methods that are de-

signed and utilized in the scope of this thesis to reduce the profiling overhead. First, scaling checks and inter- and extrapolation methods for OpenMP kernels developed for this thesis are discussed. Additionally, a method to predict the execution time of unknown OpenCL kernels based on static code analysis is designed and evaluated.

The chapter concludes with a summary and conclusion (s. Sec. 6.5), which analyzes the results and places the chapter in the scope of this thesis. Previously published content [14, 13] and contents of supervised bachelor's [87] and master's theses [88] are used in this chapter.

6.1 Introduction & Related Work

The fundamental basis for a self-adapting and self-optimizing system is knowledge of the system about itself [89, 55], called self-awareness. To gain this knowledge, a system has to monitor its own behavior and state. As systems usually are not completely separated and closed off from their surroundings, the system environment also has to be monitored. Modern hardware platforms offer a variety of performance counters to profile the behavior of a system.

Intel introduced performance monitoring since the Pentium processor with a set of model-specific performance-monitoring counter registers, called model specific registers (MSRs) [90]. These registers can be used to count several events. The special instructions `rdmsr` and `wrmsr` are utilized to read and write these registers. This can only be done by the operating system. An example that employs MSRs is Intel's Runnig Average Power Limit (RAPL) interface that exposes energy counters to the user. AMD included so called Performance Monitor Counters (PMCs) into their x-86 based processors [91]. PMCs are able to monitor various micro-architectural events in a CPU core and can be either utilized in counting or sampling mode.

NVIDIA offers a C-based API for monitoring and managing various states of the NVIDIA GPU devices called NVIDIA Management Library (NVML) [92]. NVML is part of the NVIDIA display driver. It supports querying, for example, energy and power consumption, and GPU utilization and temperature.

Different hardware platforms offer different performance counters with differing APIs to access them. To reduce this complexity for the user, there exist several libraries that offer a general abstraction for different hardware vendor APIs. With the inclusion of `perf_events` in the Linux kernel since 2009, Linux offers an interface to access hardware perfomance counters via the system call `perf_event_open()` [93]. Calling `perf_event_open()` allocates file descriptors that are associated with an event to be measured. The syscalls `ioctl` and `prctl()` are used to enable and disable events.

A well-known library that offers such an abstraction for Linux systems is lm-sensors [94]. lm-sensors provide access to health monitors like fan speeds, system voltages, and several temperatures.

Another library that provides such an abstraction is the Performance Application Programming Interface (PAPI). PAPI, developed by the University of Tennessee [95], is a user-level library that grants easy access to performance counters for heterogeneous systems. For example, CPU counters are accessed via the Linux kernel interface `perf_events`. Additionally, PAPI also offers abstractions for the lm-sensors library or NVIDIA's System Management Interface (nvidia-smi). PAPI provides developers two interfaces to instrument their application. The high level interface is simple to use and allows fast access to standard events that are present in most architectures, called preset events. In contrast, the low level interface allows more detailed control and access to so-called native events that are specific to the underlying architecture.

LIKWID [96] is a set of command line tools for Linux systems with x86 processors that are designed to support performance optimization. Within this tool set is a tool, called *likwid-perfCtr*, that grants access to and measures performace counter values [97]. Similar to PAPI, likwid-perfCtr also allows the usage of predefined and additional native events.

NVIDIA offers the CUDA Profiling Tools Interface (CUPTI) [98] to collect hardware performance counter values on NVIDIA GPUs. CUPTI provides the following APIs: the Activity API, the Callback API, the Event API, the Metric API, and the Profiler API. The CUPTI Event API allows to simply query, configure, start, stop, and read event counters on CUDA-enabled devices. Application metrics can be collected with the CUPTI Metric API Different devices have their own compute capability. The metrics available for each device are determined by the device's compute capability and can be obtained via the CUPTI Documentation [98].

In the literature, there are several other frameworks and runtime systems that use system monitoring to gain knowledge. The aforementioned TANGO framework and StarPU (s. Sec. 3.5) employ a monitoring component. TANGO utilizes its component to collect information for its event-driven adaptation manager [74]. StarPU employs a monitoring component to create performance models for task scheduling [78]. Both approaches store this monitoring data in a history database.

Similar to StarPU, the elastic computing framework [99], the task-centric runtime system by Podobas et al. [100], the scheduling framework by Jiménez et al. [101], and the performance model driven runtime by Pienaar et al. [102] create a performance database by monitoring task execution times. The performance database is then used to optimize task scheduling.

In this thesis, monitoring is not restricted to task execution times and not only used to optimize task scheduling. Rather, it is utilized to create a system state snapshot and a potential snapshot of the system environment. Combined, this allows for a dynamic and proactive adaptation of the system similar to the TANGO framework. However, the adaptation mechanisms deployed in this thesis differ significantly from the mechanisms used in the TANGO framework [77] (s. Sec. 3.4).

6.2 Monitoring System Behavior

This thesis utilizes the monitoring component of HALadapt introduced in Sec. 3.3.1. HALadapt already provides sensor plugins for the wall clock time to measure task execution times, Intel's RAPL interface to measure the energy and power consumption of modern Intel CPUs and GPUs, and the lm-sensors interface to measure the temperatures of CPU cores. As desktop and HPC systems usually do not use Intel GPUs as accelerators, an additional sensor plugin for NVIDIAs Management Library (NVML) [92] is added in this thesis. This enables HALadapt to monitor the energy and power consumption, and temperature of modern NVIDIA GPUs.

For multicore CPUs, HALadapt supports the OpenMP programming model. In its current state, HALadapt always uses all available cores when executing an OpenMP task by default. However, many existing applications are not ready for such an increased degree of parallelism and are limited in their ability to scale with an increasing number of processing cores. To detect the scaling ability of OpenMP tasks and find a fitting number of cores to map the task to, I added a new mechanism to HALadapt [16]. The mechanism, described in detail in Sec. 6.4, extends the history-based profiling mechanism to also include the number of cores as part of the database key. This enables HALadapt to associate stored characteristics with the number of cores used and hence predict execution times for varying core numbers. As the prediction mechanism employs inter- and extrapolation, a reliable prediction is only possible, if the kernel behaves in a somewhat regular pattern. A possible future extension for the profiling mechanism is to allow the developer to define problem size functions that can be used to model more irregular behavior patterns.

In the form of `intel_pstate` [103, 104], Intel provides a scaling driver for modern Intel CPUs. It is part of the CPUFreq subsystem, which supports CPU scaling, of the Linux kernel. The `intel_pstate` driver, for example, allows to set the maximum or minimum frequency as percentage of the maximum available frequency a CPU is allowed to use. By utilizing the functionality of this driver, HALadapt is able to dynamically scale the cpu core frequencies, thereby adapting the energy consumption of the core and extending the optimization potential of HALadapt. To consider frequency scaling in the history-database, the scaling factor is added as an additional database key.

Chapter 1 and Chapter 5 single out reliability as an important optimization goal of modern computing systems. However, there is no single hardware counter available that can be read to derive the current reliability of the system or a system component. The next section presents a solution in the form on an online heuristic reliability metric based on symptom-based fault detection.

6.3 A Heuristic Reliability Metric

Increasing system reliability and availability is a major concern for modern and future computing systems as mentioned before (s. Chap. 1). However, there is no performance counter that can just be accessed to derive the current reliability of the system or a system component. In order to consider reliability in the adaptation process within this thesis, a metric that reflects the reliability of a component is needed. Therefore, for this thesis I created a heuristic measure of a computing unit's reliability. The metric is computed by an online heuristic fault rate that is comprised of counting faulty execution runs and computing a ratio of faulty to correct runs in a limited window.

To create as little overhead as possible, a lightweight fault detection method is needed to implement this mechanism. A method that satisfies this requirement is symptom-based fault detection. In the following section, the faults, which shall be detected, are defined and the general concept of symptom-based fault detection and its implementation in this thesis are explained in detail.

6.3.1 Symptom-based Fault Detection

To define **faults**, **errors**, and **failures**, I use the work of Salfner et al. [105]:

- A **failure** refers to misbehavior that can be observed by the user. This means there may be something wrong inside the system, but as long as this does not result in incorrect output there is no failure.
- An **error** is defined as the deviation of the system state from the correct state. Hence, an error may lead to the service failure of a system, but also can stay unnoticed.
- **Faults** are then the hypothesized cause of an error. This means that errors are manifestations of faults.

There is a constant rising of the failure rate of today's computing systems. Especially, soft errors in hardware that are random and of temporary nature occur more often, as they are caused by lowering the system voltage in the creation of energy-efficient products [106]. These faults are particularly hard to detect as they do not always lead to wrong results and may not be reproducible. Symptom-based fault detection in this thesis is used to detect such faults.

The general concept of symptom-based fault detection is based on the following hypothesis: Systems exhibit steady-state performance behavior with few variations in the non-faulty case. However, a fault manifests itself as increasingly unstable performance-related behavior before escalating into a failure [107]. This means that a symptom for a fault occurrence manifests itself as variation of performance-related behavior, e.g., the

number of executed instructions, cache access behavior and hit rates, or branch behavior. Performance-related behavior can easily be monitored by performance counters present in modern architectures. To sum up, the basic concept is to monitor performance counters and assume the occurrence of faults if their values vary significantly compared to a baseline.

The first step to symptom-based fault detection in my approach, is the creation of a database, which stores the performance behavior of correct executions. As the performance behavior of an application is usually input-dependent, this has to be considered while creating the database. This means that a database entry is only valid for a certain class of inputs or a specific input and significantly different inputs require multiple database entries and profiling runs.

As a wide range of performance-related metrics are available, relevant metrics have to be filtered out. In this thesis, a metric is defined as relevant, if its values do not vary significantly during repeated runs without faults and show significant variance in the presence of faults. Relevant metrics can be found via profiling runs. If the profiling runs only include executions without faults, the set of possibly relevant metrics can be at least reduced to metrics that are stable during repeated executions. Additionally, a lower and an upper threshold for the values of the selected metrics have to be set. These thresholds are utilized to detect anomalies later, i.e., if a monitored value lies outside of these thresholds, the occurrence of a fault is suspected. The concept of symptom-based fault detection in this thesis is illustrated in Fig. 6.1.

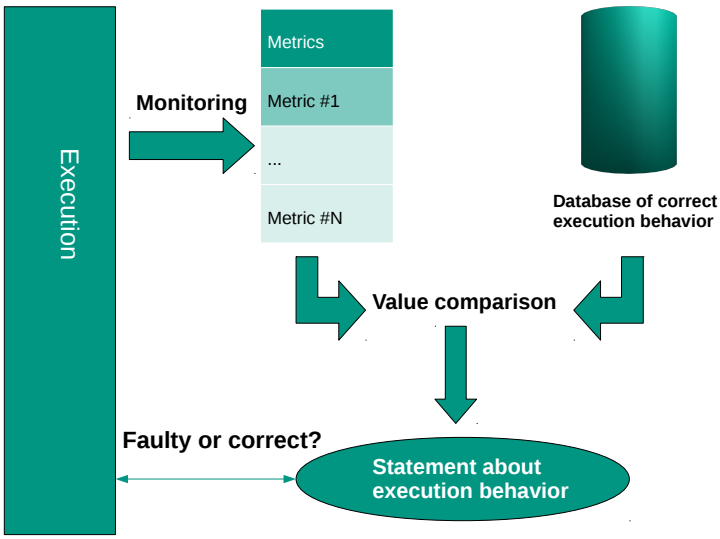


Figure 6.1: The general concept of symptom-based fault detection used in this thesis

A drawback of considering a wide spectrum of different performance metrics is that only a limited amount of hardware counters are available in today's architectures and the number of counters is dependent on the platform. Hence, multiple profiling runs may be necessary to collect all relevant performance data.

6.3.2 Related Work for Symptom-based Fault Detection

Symptom-based fault detection has be done before in the literature. Arulaj et al. [108] use performance counters as symptoms to detect concurrency bugs in production-run systems. They access the performance counters via Linux perf.

Yilmaz et al. [109] combine software-level instrumentation with hardware performance counters to associate the counters with specific function invocations and to measure execution times. They use this method to distinguish failed executions caused by software faults from successful ones.

In their work, Dimitrov et al. [110] use the detection of abnormal software behavior to predict software bugs in an effort to find the root cause of an occurred software failure. To predict bugs, they utilize software invariants learned during passing program runs, store sets for memory accesses, and the monitoring of the number of loop iterations. In contrast to the aforementioned papers, this work focuses on detecting hardware errors, in particular soft errors that are random and temporary in nature.

Williams et al. [107] employ different performance counters to detect anomalous behavior that should form patterns leading up to failures. With an anomaly detector, they create a time series that serves as input for a failure predictor. The predictor checks if there is a pattern that indicates escalating instability, which then signals an impending failure. This contrasts the approach of this thesis, as failures are not predicted but rather detected after they have occurred.

Narayanasamy et al. [111] focus on the branch predictor, the store set predictor and L2 cache accesses. They assume that a faulty execution leads to an increase in *undesirable outcomes*, e.g., a misprediction by a branch predictor.

The ReStore architecture by Wang et al. [112] uses symptom-based fault detection combined with a checkpointing mechanism. If the fault detection signals the occurrence of a fault, the architectural state of an earlier checkpoint is restored. Exceptions, branch mispredictions coupled with a confidence predictor for the branch and event logs that store events, like control instruction outcomes, are utilized as symptoms.

mSWAT [113] is a fault detection and fault diagnosis framework for multicore architectures. The framework uses a fatal-traps detector, a hang detector checking branch frequencies, a high-OS detector that monitors OS invocations, and a kernel panic detector as symptoms. If a symptom occurs, a diagnosis mechanism is invoked that decides whether the fault is just a software bug or a hardware fault, whether it is a transient or permanent fault, and which core is faulty. This is done by tracing and replaying execution.

The three aforementioned approaches focus on a specific set of symptoms to detect failures. In this thesis, there is no limitation on the set of possible symptoms and several different performance metrics are considered as potential candidates. Thereby, my approach is more general and can detect a wider range of faults.

Fault detection has also be studied on GPUs. Ding et al. [114] give an online method, which is extended from the traditional algorithm-based fault tolerance (ABFT). This method can detect faults in matrix multiplications on GPUs, then locate and correct them using row and column checksum vectors for failure detection.

Maruyama et al. [115] propose a new software framework that is able to detect Bit flips in the GPU DRAM by using a parity-based error detection code. These faults can then be recovered through checkpointing.

Carlo et al. [116] present an approach based on Software-Based-Self-Test methodology (SBST). Several parallel instances of the test kernel will be run on the GPU. The related test results (TR) or test signatures (TS) of each instance will be calculated and compared with a pre-computed Golden TR (GR)/Golden TS (GS). This approach allows to detect and locate faults on the GPU.

RISE [117] is a technique for fault detection. RISE consists of full RISE and partial RISE. Full RISE can use fully idled streaming processors in the GPU core to detect soft errors during the pipeline stall time. Partial RISE employs partially idled streaming processors to optimize reliability during branch divergence. During the idle time of the streaming processor (SP), redundant threads are executed to detect errors and improve the reliability of the SP.

Compared to these approaches, the symptom-based fault detection mechanism used in this thesis is not application-dependent and not limited to specific faults. Rather, my approach can be deployed independently of the executing application and targets the wide range of soft errors in hardware.

6.3.3 Evaluation of Symptom-based Fault Detection

As this thesis targets heterogeneous systems, symptom-based fault detection was implemented and evaluated for both CPUs [13] and GPUs [87]. Evaluating symptom-based fault detection requires the presence of faults. Therefore, fault injection is used to create faults in task executions. Fault injection is the deliberate triggering of faults with the objective to observe the resulting behavior and to test error handling code. It can be done directly in hardware or by using specific software tools. In this work, software implemented fault injection is deployed. As fault injection is an important technique in proving the correctness and robustness of a system or software, many tools and libraries exist. For this work, FINJ [118], a fault injection tool for HPC systems, and SASSIFI [119, 120], a fault injection framework for NVIDIA GPUs, were chosen. FINJ is implemented in Python and based upon tasks. Thereby, a task can represent a benchmark or a

fault-triggering program. The execution of a workload of tasks is controlled by a specific controller that schedules and starts the tasks on an engine. SASSIFI builds on top of SASSI [121], which is a low-level assembly-language instrumentation tool that provides the ability to instrument instructions in the low-level GPU assembly language (SASS).

For the evaluation, a selected number of benchmarks are executed and injected with faults. For each run, only one specific fault is used. If there are monitored metrics whose values lie outside of the chosen thresholds, the injected fault is assumed to be detected. After all experiments are conducted, an analysis step follows. First, the injected faults are classified. For example, all faults affecting memory are grouped together. Then, the monitored results are checked if faults belonging to the same class show similar changes in the runtime metrics. It is also checked, if faults belonging to different classes can be differentiated by observing the monitored runtime behavior.

The next paragraph introduces the faults injected in the experiments. After that, the benchmarks used for evaluation are presented. In the third paragraph, the experimental setup is explained. Then, the results are presented and discussed.

Injected Faults

In this work, three types of faults are analyzed on the CPU: the alteration of loop index variables to create random memory accesses, the reduction of loop iterations, and interferences created by the FINJ library. The interferences are used to mimic anomalies in real-life systems by stressing single components, emulating interference or malfunction in that component.

The alteration of loop index variables is done by overwriting the current value of the index variable in the pages of the process in main memory via opening */proc/$procid/mem/* and then jumping to the address of the variable. An example can be seen in Listing 6.1.

```
1    FILE *mem = fopen("/proc/$procid/mem/", "w");
2    fseek(mem, (uintptr_t) &i, SEEK_CUR);
3    fwrite(&manipulation, sizeof(i), 1, mem);
4    fclose(mem);
```

Listing 6.1: Altering an index variable

The process id and and the variable address can be obtained by calling `popen("pid of $processname", "r")` and writing out the address to a file that can be read by the alteration process, respectively. As I only want to create random accesses, I made sure that the altered index value always lies within the given range and that the correct number of iterations is executed. In the same way, the number of iterations can be altered by overwriting the current iteration bound. From the FINJ library five interference applications inspired by Tuncer et al. [122] are used:

- `copy`: Creates interferences on the hard disc drive (HDD) by constant file in- and output

- `ddot:` Creates interferences on the arithmetic logic unit (ALU) by constantly executing a floating-point matrix multiplication
- `dial:` Creates interferences on the ALU by constantly executing floating-point math instructions
- `leak:` Creates a controlled memory leak by constantly allocating new arrays and calling `memcpy()`
- `memeater:` Creates a controlled memory leak by constantly (re-)allocating an array and executing integer additions

For faults on GPUs, SASSIFI provides an automated framework to perform error injection campaigns and can be used to perform many types of resilience evaluation studies. It offers three fault injection modes.

- IOV (Instruction Output Value) mode: Using this mode, faults can be injected into the destination register values.
- IOA (Instruction Output Address) mode: With this mode, faults can be injected into destination register indices and store addresses.
- RF (Register File) mode: This mode allows the injection of faults in the Register File.

SASSI, the instrumentation tool SASSIFI is built on, offers four compilation flags that can be utilized for fault injection: **–sassi-inst-before**, **–sassi-before-args**, **–sassi-inst-after**, and **–sassi-after-args**. These flags allow the insertion of operations necessary for fault injection before or after the execution of selected instructions.

Fig. 6.2 shows the operations that need to be inserted for fault injection when deploying different modes. For example, if we select the IOV mode, all operations are inserted after the instruction is executed.

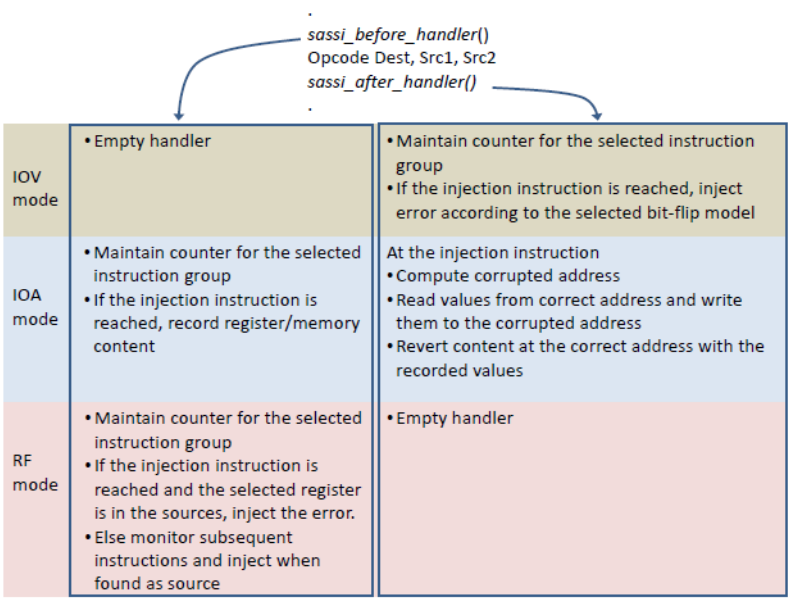


Figure 6.2: Error injection handlers for different modes [119].

In different modes different instruction groups can be selected for fault injection. Fig. 6.3 shows the instruction groups that can be selected for each mode and the faults that can be injected.

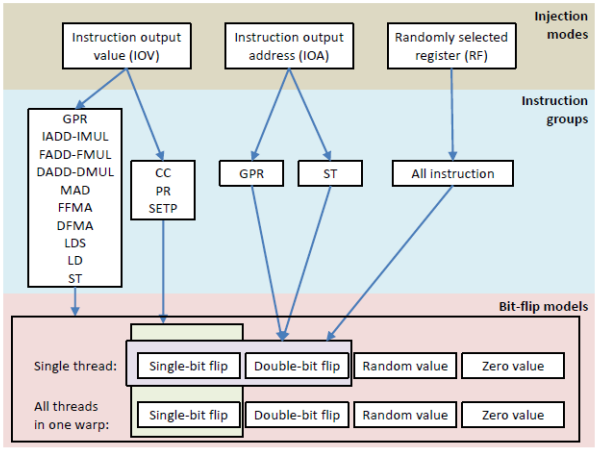


Figure 6.3: Summary of different error injection modes, instruction groups, and bit-flip models (BFM) that SASSIFI provides [119].

In total, the following instruction groups (IG) can be selected:

- Instructions that write to general purpose registers (GPR)

- Instructions that write to condition code (CC)
- Instructions that write to a predicate register (PR)
- Store instructions (ST)
- Integer add and multiply instructions (IADD-IMAD-OP)
- Single precision floating point add and multiply instructions (FADD-FMUL-OP)
- Double precision floating point add and multiply instructions (DFADD-DFMUL-OP)
- Integer fused multiply and add (MAD) instructions (MAD-OP)
- Single precision floating point fused multiply and add (FMA) instructions (FMA-OP)
- Double precision floating point fused multiply and add (DFMA) instructions (DFMA-OP)
- Instructions that compare source registers and set a predicate register (SETP-OP)
- Loads from shared memory (LDS-OP)
- Load instructions, excluding LDS instructions (LD-OP)

Before performing the fault injection task, SASSIFI had to be modified as it included randomization and the experiments had to be deterministic and repeatable. This can be done in the file */sassifi/scripts/generate_injection_list.py*. In this file, we can find the following code:

```
1             def write_injection_list_file(app, inj_mode, igid, bfm,
                  num_injections, total_count, countList):
2                ...
3                ...
4                while num_injections > 0 and total_count != 0: # first two are
                      kname and kcount
5                   num_injections -= 1
6                   injection_num = random.randint(0, total_count) # randomly
                        select an injection index
7                   if igid == "rf":
8                      ...
9                   else:
10                     ...
11                     inj_op_id_seed = random.random()
12                  inj_bid_seed = random.random()
13                  ...
14               ...
```

Here, to determine the injection fault, three variables, *injection_num*, *inj_op_id_seed*, and *inj_bid_seed*, have to be modified and their randomization removed.

Following experiments showed that fault injection with SASSIFI always causes an increase in the number of executed instructions, and these additional instructions are not caused by the injected fault, but are used to generate the fault. These additional instructions also cause unavoidable deviations in other metrics. Therefore, it cannot be distinguished whether the deviation of other metrics is caused by the injected fault or these additional instructions. In order to solve this problem, the deviations caused by these additional instructions need to be eliminated. The file */SASSI/instlibs/src/err_injector/injector.cu* contains the code to perform fault injection. The code for fault injection is as follows:

```
1            ...
2            print_dest_reg_values(cp, rp, "before");
3            // get the value in the register, and inject error
4            int32_t valueInReg = rp->GetRegValue(cp, regInfo).asInt;
5
6            ...
7            SASSIRegisterParams::GPRRegValue injectedVal;
8            injectedVal.asUint = 0;
9            uint32_t injBID = 0;
10           if (injBFM == FLIP_SINGLE_BIT || injBFM == WARP_FLIP_SINGLE_BIT)
                 {
11              injBID = get_int_inj_id(32, injBIDSeed);
12              injectedVal.asUint = valueInReg ^ (1<<injBID); // actual error
                     injection
13           } else if (injBFM == FLIP_TWO_BITS || injBFM ==
                 WARP_FLIP_TWO_BITS) {
14              injBID = get_int_inj_id(31, injBIDSeed);
15              injectedVal.asUint = valueInReg ^ (3<<injBID); // actual error
                     injection
16           } else if (injBFM == RANDOM_VALUE || injBFM == WARP_RANDOM_VALUE)
                  {
17              injectedVal.asUint = ((uint32_t)-1) * injBIDSeed;
18           } else if (injBFM == ZERO_VALUE || injBFM == WARP_ZERO_VALUE) {
19              injectedVal.asUint = 0;
20           }
21
22           ...
23           if (!DUMMY_INJECTION) {
24              rp->SetRegValue(cp, regInfo, injectedVal);
25           }
26
27           int32_t valueInRegAfter = rp->GetRegValue(cp, regInfo).asInt;
28           ...
29           print_dest_reg_values(cp, rp, "after");
```

The process of fault injection in IOV mode is as follows. First, get the original value in the register and save it in *valueInReg*. Then, according to different fault models, the new value to be written to the register will be obtained and set into *injectedVal*. If the single-bit-flip model or the double-bits-flip model is selected, then a bitwise XOR will be performed on *valueInReg*. If the random value model or zero value model is selected, then the new value will be set directly. Finally, the new value saved in *injectedVal* is written into the register by calling the function *SetRegValue()* to complete the fault injection.

If the value rewritten to the register during the fault injection is the original value in the register, then it is equivalent to no fault being injected. The modified code is as follows:

```
1            ...
2            SASSIRegisterParams::GPRRegValue injectedVal;
3            injectedVal.asUint = 0;
4            uint32_t injBID = 0;
5            if (injBFM == FLIP_SINGLE_BIT || injBFM == WARP_FLIP_SINGLE_BIT)
                 {
6               injBID = get_int_inj_id(32, injBIDSeed);
7               injectedVal.asUint = valueInReg | (0<<injBID) ; // actual
                    error injection
8            } else if (injBFM == FLIP_TWO_BITS || injBFM ==
                 WARP_FLIP_TWO_BITS) {
9               injBID = get_int_inj_id(31, injBIDSeed);
10              injectedVal.asUint = valueInReg | (0<<injBID) ; // actual
                    error injection
11           } else if (injBFM == RANDOM_VALUE || injBFM == WARP_RANDOM_VALUE)
                   {
12              injectedVal.asUint = valueInReg ;
13           } else if (injBFM == ZERO_VALUE || injBFM == WARP_ZERO_VALUE) {
14              injectedVal.asUint = valueInReg ;
15           }
```

In order to reduce the impact of code changes on the results as much as possible, when the single-bit-flip model or double-bits-flip model is selected, a bitwise OR with zero will additionally be performed. This code is then utilized to run applications without faults to fill the database. Thus, the application runs without faults also execute the additional instructions necessary to inject faults. Since SASSI [121] is closed source, the specific code of the function *SetRegValue()* and its effects are unkown. After experiments, it was found that the alteration does not completely eliminate all additional instructions. However, additional instructions could be minimized and useful baseline profiling runs could be conducted.

For the actual fault injection, IOV mode was chosen as IOV supports eight bit-flip models. The following three models that are implemented in the current version of SASSIFI were chosen for the evaluation: Double bit-flip, Random value, and Zero value:

- Double bit-flip: bit-flips in two adjacent bits in one register in one thread
- Random value: random value in one register in one thread
- Zero value: zero out the value of one register in one thread

As instruction groups, we selected GPR, FADD-FMUL-OP, and LDS-OP.

Benchmarks

As a first benchmark, I implemented a floating-point matrix multiplication (**mMult**) with 300×300 and 500×500 matrices initialized with random floating-point numbers. The other benchmarks used are taken from the Rodinia Benchmark Suite [123]:

Hotspot3D iteratively computes the heat distribution of a 3d chip represented by a grid. In every iteration, a new temperature value depending on the last value, the surrounding values, and a power value is computed for each element. For the CPU evaluation, I utilized a $512 \times 512 \times 8$ grid with the start values for temperature and power included in the benchmark suite, and a total of 1000 iterations.

Hotspot is the 2D variant of Hotspot3D. A 64×64 grid with a pyramid height 2 and two iterations serves as evaluation data for the GPU experiment. Again, the input files provided by the benchmark suite are used.

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations used to remove noise without destroying important image features. The benchmark consists of image extraction, continuous iterations over the image, and image compression. As input, I used the 502×458 image provided by the benchmark suite with 100 iterations and $\lambda = 0.5$.

Nearest Neighbor (NN) is used to finds the k-nearest neighbors from an unstructured data set. It first reads the record, then calculates the Euclidean distance from the given target, and finally gets the k nearest neighbors. The default inputs of the Rodinia Benchmark Suite are utilized and four nearest neighbors are outputted.

These benchmarks were selected because they represent typical workloads with various types of parallelism and data access patterns. They also specifically target heterogeneous architectures and therefore are available in the programming models OpenMP, OpenCL, and CUDA.

Experimental Setup

For the experiments, PAPI or CUPTI instrumentation code was added to each benchmark described above in order to monitor selected performance counters, respectively. All experiments are executed ten times with and without fault injection, respectively. Fig. 6.4 displays the experimental concept used for the CPU experiments. The results show the average $\varnothing$ and the standard variation s of the specific metric measured over all ten

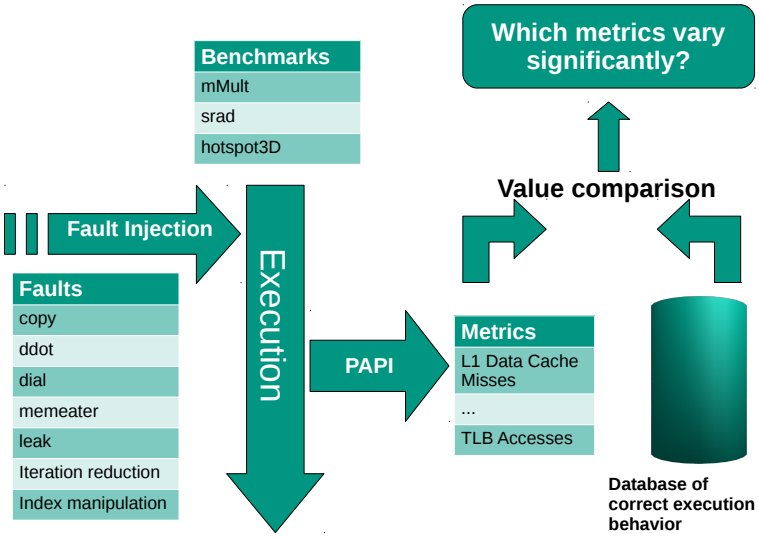


Figure 6.4: The general concept of the symptom-based fault detection evaluation process on the CPU.

executions with and without the injected fault, respectively. Additionally, I compute an occurrence ratio, that shows how often the value of the measured metric varied significantly from the non-faulty case. This means if a value increases significantly in 6 out of 10 execution runs, the occurrence ratio would be $60\,\%$.

The experiments are conducted on a server with two Intel Xeon E5-2650 v4 CPUs a 12 cores each, an NVIDIA Tesla K80, and 128 GB DDR4 SDRAM DIMM (PC4-19200). The software environment includes Ubuntu 18.04.1, the Linux 4.15.0-43-generic kernel, glibc 2.27, and CUDA 7.0, as SASSIFI does not work with newer CUDA versions.

Results

The next paragraphs present the **CPU experiments** that consist of the loop alteration, instruction number reduction, and FINJ's interference applications. The experiments are sorted by the injected fault.

The alteration of the loop index variable creates random accesses into the used data structure. This changes the data cache behavior of the benchmark increasing misses as the random accesses violate the locality principle. The results of the mMult benchmark (s. Table 6.1) show these changes. Misses in the data translation lookaside buffer (TLB DM) and in the L2 data cache (L2 DCM), data prefetch misses (PRF DM), and L3 data cache accesses (L3 DCA) increase significantly.

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI TLB DM	27.8	13.8	1882	258.7	$100\,\%$
PAPI PRF DM	88898.7	290.1	503561.3	794	$100\,\%$
PAPI L2 DCM	15733227	1125.2	16188659.3	4214.8	$100\,\%$
PAPI L3 DCA	15733439.3	312.3	16185326.6	1193.8	$100\,\%$

Table 6.1: Results of the combination of mMult and the manipulation of the loop index

The reduction of the number of iterations effectively leads to a reduction of issued and executed instructions. In general, the instruction performance counters are very precise, e.g., the counters for the executed floating-point operations always match the actually executed operations with a deviation of 0. Therefore, a reduction of the executed instructions is easily recognizable using the provided instruction counters as can be seen exemplary in the results of the mMult benchmark in Table 6.2. Here, the total floating-point operations, the floating-point additions, and floating-point multiplications scale according to the number of executed loops. As these results are pretty straightforward, the results of the other two benchmarks are omitted.

Symptom	mMult w/o faults	mMult w faults		
	N = 300	N = 200	N = 100	N = 50
PAPI FP OPS	54 $\cdot 10^6$	36 $\cdot 10^6$	18 $\cdot 10^6$	9 $\cdot 10^6$
PAPI FML OPS	27 $\cdot 10^6$	18 $\cdot 10^6$	9 $\cdot 10^6$	4.5 $\cdot 10^6$
PAPI FADD OPS	27 $\cdot 10^6$	18 $\cdot 10^6$	9 $\cdot 10^6$	4.5 $\cdot 10^6$

Table 6.2: Results of the iteration number reduction for the matrix multiplication benchmark

Copy creates file I/O overhead and thereby simulates HDD interferences. Expected results are variations in the lower-level cache structures and the TLB. These expectations are confirmed by the results of the SRAD benchmark shown in Table 6.3. In 9 out of 10 runs, large variations can be noted for the L3 total cache misses (L3 TCM) and TLB DM counters. Additionally, the total number of stalls increases by about $3\,\%$ on average and the number of L2 instruction cache misses (ICM) by about $150\,\%$ on average. However, these two symptoms occur less often and in the case of the L2 ICMs vary significantly between runs, which aggravates a detection. The results for mMult (s. Table 6.4) show a similarity to the results of SRAD where the PRF DMs and total stalls increase, but the biggest variations are seen in the instruction caches. Hotspot3D has similar results with

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurrence ratio
PAPI L3 TCM	1.8	1.87	44.44	54.81	$90\,\%$
PAPI TLB DM	11899.7	1080.84	19816.7	3421.12	$90\,\%$
STALLS TOTAL	79803752.8	155218.56	82099161.3	1197866.62	$80\,\%$
PAPI L2 ICM	2088	154.6	3188.3	1366.39	$60\,\%$

Table 6.3: Results of the combination of SRAD and copy

an increase in L2 and L3 cache accesses as well as symptoms such as an increase in L3 data cache writes (DCW) and in cycles stalled waiting for memory writes (PAPI MEM WCY).

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICA	92.7	8.06	751.3	29.04	$100\,\%$
PAPI L3 ICA	110.1	26.76	227	20.33	$100\,\%$
PAPI PRF DM	258091.5	2350.31	270598.5	1349.06	$100\,\%$
STALLS TOTAL	33806710.3	448643.96	35368628.1	1226479.31	$80\,\%$

Table 6.4: Results of the combination of the mMult and copy

Leak leads to data cache and TLB misses via a controlled memory leak. The results of SRAD in Table 6.5 show that the number of L3 TCMs and the number of total cycles increase significantly throughout all test runs. Additional symptoms are TLB DMs and L3 instruction caches accesses (L3 ICA), which are visible in $80\,\%$ of the conducted executions. Similar to the copy benchmark, variations for Hotspot3D are mostly visible in

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occ. rat.
PAPI L3 TCM	1.8	1.87	11061.6	17445.73	$100\,\%$
PAPI REF CYC	911683861	4293565.15	1042523457.8	18936885.44	$100\,\%$
PAPI TLB DM	11899.7	1080.84	16347.5	4986.21	$80\,\%$
PAPI L3 ICA	1950.6	101.63	2439.8	262.8	$80\,\%$

Table 6.5: Results of the combination of SRAD and leak

the instruction caches (s. Table 6.6). In this case however, there is no single symptom present in all test runs. mMult also only shows two symptoms in combination with leak: increases in total stalls and TLB IMs, which are visible in $80\,\%$ of the test runs.

Symptom	Hotspot3D w/o faults		Hotspot3D w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICM	1360	269.32	1870.1	252.18	$80\,\%$
PAPI L1 ICM	1641.3	251.18	2079.1	358.2	$70\,\%$

Table 6.6: Results of the combination of Hotspot3D and leak

Memeater affects the system similarly to leak. Additionally, memeater creates misses in the instruction caches. Table 6.7 presents the observed symptoms for SRAD. Mirroring the results of leak, the number of total cache misses and the total cycle number increase significantly in all test runs. TLB data misses are also significantly augmented again and observable in eight executions. Furthermore, the additional instructions lead to an increase in L2 ICMs. The results for the combination of Hotspot3D and memeater are

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurr. ratio
PAPI L3 TCM	1.8	1.87	82869	133062.97	$100\,\%$
PAPI REF CYC	911683861	4293565.15	1052188361.2	19103730.87	$100\,\%$
PAPI TLB DM	11899.7	1080.84	18344.5	6625.07	$80\,\%$
PAPI L2 ICM	2088	154.6	2524.5	333.03	$80\,\%$

Table 6.7: Results of the combination of SRAD and memeater

identical to the combination of Hotspot3D and leak with increases in L1 and L2 ICMs and MEM WCYs. Even the occurrence ratio is identical for all three symptoms. For mMult, the results resemble the results of the copy benchmark instead of leak, as the visible symptoms are increases in L1 and L2 ICMs, total stalls, PRF DMs, and additionally an increase in cycles with maximum instruction issue (FUL ICY).

Dial does not use much additional data, so mostly variations in the instruction caches are expected. For SRAD, the results are displayed in Table 6.8. As expected, significant increases in the L2 ICMs are measured. These correlate with the decrease in instructions cache hits (ICH), and an increase in L3 instruction cache accesses (ICA) (and reads (ICR)). All those symptoms are observable in $100\,\%$ of the execution runs. Contrary to expectations an increase in TLB DMs and a decrease in L2 DCAs is visible. Symptoms for mMult also manifest themselves in an increase in ICMs and correlating increases in L2 and L3 ICAs. Furthermore, the number of prefetch data misses (PRF DM) decreases in every run. Hotspot3D also showed significant increases (up to $400\,\%$) in instruction cache misses. Surprisingly, I measured a decrease in TLB DMs and MEM WCYs.

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurr. ratio
PAPI L2 ICM	2088	154.6	2915.9	162.82	100 %
PAPI L2 ICH	20175.3	512.32	18633	169.35	100 %
PAPI L3 ICA	1950.6	101.63	2956.6	92.42	100 %
PAPI TLB DM	11899.7	1080.84	15885.1	2255.44	100 %
PAPI L2 DCA	19876232.64	2957162.47	10991900.1	14624.57	90 %

Table 6.8: Results of the combination of SRAD and dial

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI PRF DM	258091.5	2350.3	236171	15761.98	100 %
PAPI L1 ICM	109.8	16.7	170.5	25.02	90 %
PAPI L2 ICM	130.5	19.34	155.9	14.92	70 %
PAPI L3 ICA	93.4	6.6	130.4	21.98	70 %

Table 6.9: Results of the combination of mMult and dial

Ddot uses more additional data in the form of matrices compared to dial. The results for SRAD (s. Table 6.10) show the effects of the additional instructions executed on the instruction caches. Again, the ICMs on the L2 level increase, which correlates with the increase of L3 ICAs (and ICRs) and decrease of L2 ICHs. The data usage is not really visible in the monitored values, as the only visible variation was a decrease in L2 data cache accesses, which is also visible for SRAD with dial. Similar to the results for dial,

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICM	2088	154.6	2784.3	114.36	100 %
PAPI L3 ICA	1950.6	101.63	2830.9	67.07	100 %
PAPI L2 DCA	19517962.78	3226987.33	11026045.4	66239.26	90 %
PAPI L2 ICH	20175.3	512.32	18740.4	445.68	90 %

Table 6.10: Results of the combination of SRAD and ddot

significant increase in L1 and L2 ICMs for Hotspot3D is observed. Additionally, an increase in misses in the instruction TLB (ITLB) is noticed. Again, the number of TLB data misses decrease compared to the execution without interferences. The mMult benchmark also shows similar results to dial. Increases in ICMs, TLB DMs, and a decrease in PRF DMs are again detected. Additionally, increases in ITLB misses and total stalls are measured.

Symptom	Hotspot3D w/o faults		Hotspot3D w faults		
	∅	s	∅	s	occurrence ratio
PAPI L1 ICM	1277.4	208.14	5060.8	286.99	100 %
PAPI L2 ICM	1416.9	249.7	5228.9	268.77	100 %
PAPI TLB DM	513562.6	137417.52	213266.5	4514.72	90 %
ITLB MISS	587.4	257.26	958.2	312.8	70 %

Table 6.11: Results of the combination of Hotspot3D and ddot

The following paragraphs introduce the results obtained by the **GPU experiments** using SASSIFI's fault injection. For the GPU experiments, the experiments are sorted by the instruction group faults were injected into.

GPR groups instructions that write to general purpose registers. Here, injecting faults could potentially lead to changes in the cache access behavior, the number of bank conflicts, and the number of address divergences in load/store operations. As bank cache misses, bank conflicts, and address divergences may potentially lead to instruction replays, the act of issuing an instruction multiple times because it could not be completed, an alteration in the number of instructions issued is also expected. The following two tables show the results obtained when a double-bit-flip occurred. The results for SRAD (s. Tab 6.12) show the effect of the double bits flip on the issued instructions with a decrease of the number of control flow instructions issued (*cf_issued* and an increase in stalls while fetching the next instruction (*stall_inst_fetched*).

Table 6.12: The result of the combination of SRAD and Double-bit-flip (GPR)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361606.9	270.35	360419.6	304.42	100 %
stall_inst_fetch	2.00 %	0.03 %	2.10 %	0.03 %	100 %

Contrary to these results, a decrease in *cf_issued*, and increases in total instructions issued (*inst_issued*) and accordingly in total issue slots used (*issue_slots*) can be observed from the results for NN (Table 6.13). In addition, increases in total number of global memory atomic transactions (*atomic_transactions*), load/store instructions issued (*ldst_issued*), local load/store transactions (*local_load_transactions*, *local_store_transactions*), and memory read transactions seen at the L2 cache (*l2_read_transactions*) can also be observed. As only one symptom, similar to NN an increase in *cf_issued*, was visible for Hotspot, the table is omitted.

The results for the random value fault model can be seen in Table 6.14 (SRAD) and Table 6.15 (NN). Again, the table for Hotspot is omitted as the only symptom visible was

Table 6.13: The result of the combination of NN and Double-bit-flip (GPR)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r
atomic_trans.	8017793.5	73551.37	8083246.1	38972.16	50 %
cf_issued	443771.1	786.21	439837.6	547.17	100 %
inst_issued	16183137.2	84432.61	16281498.5	73144.19	70 %
issue_slots	15217601.7	84678.09	15317475.3	72717.37	70 %
l2_read_trans.	2145784.3	2458.37	2148287.9	2349.15	70 %
ldst_issued	13131635.7	47916.17	13313842.9	93257.62	90 %
local_load_trans.	750554.9	7648.26	760566.9	5937.72	70 %
local_store_trans.	3195316.8	14191.03	3224604.5	31197.69	60 %

the increase in *cf_issued*.

Table 6.14: The result of the combination of SRAD and Random Value (GPR)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361631.6	353.13	360408.8	251.32	100 %
inst_issued	10375528.1	53667.4	10329675.6	28380.96	60 %
issue_slots	9310575.8	53493.82	9265840.2	28176.83	60 %
ldst_issued	7693273.9	50224.71	7641020.6	20940.56	70 %
local_load_trans.	1836688.4	19479.34	1818154.8	14904.33	60 %
local_store_trans.	4687735.6	34407.02	4658019.4	15046.69	60 %
stall_inst_fetch	2.00 %	0.02 %	2.11 %	0.03 %	100 %

Similar to the results for the double-bit-flip model, SRAD shows the decrease *cf_issued* and the rise of *stall_inst_fetch*. Further symtpoms are the decrease of *inst_issued*, *issue_slots*, *ldst_issued*, *local_load_transactions*, and *local_store_transactions* that match the trend shown by the decrease of *cf_issued*. The results of NN are similar to the results of the previous experiment. In Table 6.15, the increase of *atomic_transactions* and the decrease of *cf_issued* can still be seen. Similarly, the increase in *inst_issued*, *ldst_issued* and *local_store_transactions* have also been observed. In addition, an increase in the device memory read transactions (*dram_read_transactions*) is observed that matches the other increases.

The experiments with the zero value fault model result in similar symptoms. The results for Hotspot are the same as before, from Table 6.17 only the increase in *cf_issued* can be seen. For SRAD, the results obtained are the same as with the double bit flip model. The decrease in *cf_issued* and the rise in *stall_inst_fetch* are visible as symptoms (the table is omitted to increase visibility). The results for NN (Table 6.16) are still similar to before. The decrease in *cf_issued*, the increase in *atomic_transactions*, *inst_issued*, *issue_slots*, *ldst_issued* and *local_store_transactions* are all observed.

Table 6.15: The result of the combination of NN and Random Value (GPR)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
atomic_trans.	7984352.7	53209.19	8100949.3	50292.49	80 %
cf_issued	444063	669.81	440123.5	341.81	100 %
dram_read_transactions	432064.6	2021.90	434797.1	2342.21	60 %
inst_issued	16130690.4	127652.67	16252896.2	114521.88	50 %
ldst_issued	13085069.7	107864.52	13289426.5	50201.94	100 %
local_store_trans.	3184418.8	26533.3	3230818.8	27115.91	80 %
stall_not_selected	14.13 %	0.18 %	13.94 %	0.16 %	60 %

FADD-FMUL-OP is the group term for single precision floating point add and multiply instructions. Again, injecting faults in this instruction group should lead to alterations in the instruction replay behavior of the benchmarks.

Table 6.16: The result of the combination of NN and Zero Value (GPR)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
atomic_trans.	8012331.3	82300.50	8142256	75547.25	80 %
cf_issued	479149.9	680.4	475359.4	498.31	100 %
inst_issued	16462281.6	117116.96	16628205.2	125970.92	70 %
issue_slots	15498139.9	115728.42	15663119.3	124161.25	70 %
ldst_issued	13202430.4	116908.54	13348558.9	141785.43	60 %
local_store_trans.	3182047.9	22732.57	3232043	26385.39	80 %

Table 6.17: The result of the combination of Hotspot and Zero Value (GPR)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	717639.5	857.74	718693.6	825.75	70 %

The following three tables, Table 6.18 for SRAD, Table 6.19 for NN, and Table 6.20 for Hotspot, show the symptoms that were visible when the double-bit-flip is injected as a fault. Similar to the results for the GPR instruction group, the changes in *cf_issued* are visible for all three benchmarks. The SRAD results also show the increase in *stall_inst_fetch* seen in the same experiment with GPR. The same trend is present in the results for NN as the increase in *inst_issued*, *issue_slots*, *ldst_issued*, and *local_store_transactions* are again observable.

Table 6.18: The result of the combination of SRAD and Double-bit-flip (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361511.4	320.63	360468.9	95.91	100 %
stall_inst_fetch	2.01 %	0.02 %	2.10 %	0.02 %	100 %

Table 6.19: The result of the combination of NN and Double-bit-flip (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	798753.4	686.81	793718.5	781.04	100 %
inst_issued	18414897.2	132823.42	18546503.7	92439.27	60 %
issue_slots	17290058.3	133262.57	17421705.3	93746.96	60 %
ldst_issued	14411021	153761.13	14579835.5	134927.8	60 %
local_store_trans.	3089973.1	28823.64	3128223.1	18492.65	80 %
stall_memory_dependency	11.81 %	0.13 %	11.66 %	0.13 %	60 %

Additionally, a decrease in the total number of stalls produced because a memory operation could not be performed (*stall_memory_dependency*) occurs. For Hotspot, additional symptons in the increase in number of read requests at the L2 cache originating from L1 (*l2_l1_read_transactions*) and an increase in *atomic_transactions* are observable.

Both the increase in *cf_issued* and *stall_inst_fetchd* for SRAD can again be seen when the random value model is used (s. Table 6.21). With an increase in *l2_read_transactions*, an additional symptom manifests itself. The results for Hotspot (s. Table 6.23) show that only *cf_issued* increased. The results for NN (s. Table 6.22) show that the fault injection still caused more instruction replays and more cache and memory accesses as *inst_issued*, *issue_slots*, *atomic_transactions*, *local_load_transactions*, and *l2_l1_write_transactions* all increased. In addition, the change in *cf_issued* can still be seen.

Table 6.20: The result of the combination of Hotspot and Double-bit-flip (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
atomic_trans.	5542783.2	27655.33	5591960.4	39260.83	70 %
cf_issued	1181101.9	499.63	1182599.9	562.46	100 %
l2_l1_read_trans.	2852582.2	861.59	2853549.4	903.69	70 %

Table 6.21: The result of the combination of SRAD and Random Value (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361669.8	210.71	360229.9	411.76	%
l2_read_trans.	2596389.8	497.96	2597172	934.48	%
stall_inst_fetch	2.00 %	0.03 %	2.08 %	0.02 %	%

Table 6.22: The result of the combination of NN and Random Value (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
atomic_trans.	9072448.2	65945.8	9165003.4	54580.97	80 %
cf_issued	798586.1	750.19	794446.1	649.88	100 %
inst_issued	18389608.4	114717.89	18521390.3	123292.32	70 %
issue_slots	17264485.4	114354.67	17395045.7	124025.01	70 %
l2_l1_write_trans.	3459754.3	2446.75	3462802.8	3332.82	60 %
local_load_trans.	734801.1	6418.04	745525.4	8203.62	80 %

The results for the experiments using the zero value model can be viewed in Table 6.24 for SRAD, Table 6.26 for Hotspot, and Table 6.25 for NN. For SRAD, the changes of *cf_issued* and *stall_inst_fetch* can still be observed. The Hotspot results show that the fault increases *dram_write_transactions* and, again, *cf_issued*. From the results for NN (Table 6.25), we can find that the decrease in *cf_issued*, the increase in *atomic_transactions*, *inst_issued*, *issue_slots*, and *local_store_transactions* are also detected again.

LDS-OP represents loads from shared memory. Injecting faults in load instructions should, again, lead to alterations in the instruction replay behavior. Additionally, the faults should also be more directly visible in metrics related to memory access instructions like load instructions issued etc. As the benchmark NN does not have instructions from the instruction group LDS-OP, no results could be produced for this benchmark.

As in the last experiments, the changes in *cf_issued* are still visible for the two benchmarks over all fault models for the instruction group LDS-OP. For SRAD (s. Table 6.27,

Table 6.23: The result of the combination of Hotspot and Random Value (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	1181169.9	575.8786	1182212.6	361.79	90 %

Table 6.24: The result of the combination of SRAD and Zero Value (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361551.6	282.14	360393.4	224.25	100 %
stall_inst_fetch	2.00 %	0.03 %	2.10 %	0.02 %	100 %

Table 6.25: The result of the combination of NN and Zero Value (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
atomic_trans.	9054403.3	69633.94	9134497.7	69987.69	70 %
cf_issued	837222.2	655.85	832885	768.72	100 %
inst_issued	18589451.7	106404.19	18778406.2	106617.53	70 %
issue_slots	17464653.3	107344.99	17654145.8	106476.13	70 %
local_store_trans.	3097506	21596.99	3123829.9	17122.16	70 %

we additionally can monitor an increase in *stall_inst_fetch*, *ldst_issued*, and *local_store_transactions*. In the results for Hotspot (s. Table 6.28) next to the increase in *cf_issued*, increases in *dram_write_transactions* and *l2_read_transactions*, and the decrease in *local_load_transactions* are detected.

The random value fault model experiment for SRAD (s. Table 6.29) results in an additonal rise of *stall_inst_fetch* that has been visible in all past experiments. For Hotspot (Table 6.30), we can observe a decrease in *sysmem_write_transactions* next to the increase in *cf_issued* for the random value fault model.

The results for the zero value fault model experiments can be seen in Table 6.31 and Table 6.32. For SRAD, changes in *cf_issued* and *stall_inst_fetch* are still observed, and the increase in *local_store_transactions* is also found. The increases in *cf_issued*, *dram_write_transactions*, and *l2_l1_read_transactions* that were observable in the double-bit-flip fault model experiment with Hotspot can also be seen here.

Table 6.26: The result of the combination of Hotspot and Zero Value (FADD-FMUL-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	1237785.1	447.97	1238655.8	456.15	80 %
dram_write_trans.	7728748.7	1564.2	7731094.7	2582.94	60 %

Table 6.27: The result of the combination of SRAD and Double-bit-flip (LDS-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361614.8	198.83	360455	244.21	100 %
ldst_issued	7664055.3	39923.27	7709413.5	33343.45	60 %
local_store_trans.	4675417.3	24104.77	4699637.3	19344.95	70 %
stall_inst_fetch	1.99 %	0.02 %	2.10 %	0.01 %	100 %

Table 6.28: The result of the combination of Hotspot and Double-bit-flip (LDS-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	1181284.1	544.89	1182223.5	621.39	70 %
dram_write_trans.	7714109.1	1969.06	7716996.7	1763.21	70 %
l2_read_trans.	4514577.1	1021.33	4516698.7	1095.79	80 %
local_load_trans.	1079206.8	3963.4	1074556.4	3690.92	70 %

Statistical Analysis

To test the statistical significance of the obtained results, Welch's t-test [124], a statistical test that is used to test the hypothesis that two means belong to the same population, is employed. If the hypothesis is accepted, the occurred symptom originates from correct behavior and not a fault. However, if the hypthesis can be discarded with a high probability, it is highly likely that the symptom does not originate from normal execution behavior. Exemplary, the results of three tests for the CPU experiments are shown here. Tables 6.33, 6.34 and 6.35 show the results for the combination of SRAD and dial, Hotspot3D and leak, and mMult and copy. For all symptoms registered in these benchmarks, the deviation that occurred in the fault-injection runs has a probability to occur in normal runs of less than $1\,\%$ and in most cases even less than $0.1\,\%$. This means that it is almost definite that the monitored symptoms do not result from the distribution observed in the fault-free runs. Therefore, it is reasonable to say that the injected faults changed the application behavior. For all conducted benchmarks, the maximum probability for a symptom occurring in a fault-free run is $3.6\,\%$. For 18 of 30 symptoms examined, the

Table 6.29: The result of the combination of SRAD and Random Value (LDS-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361600.5	339.12	360667.3	182.96	100 %
stall_inst_fetch	1.99 %	0.02 %	2.09 %	0.02 %	100 %

Table 6.30: The result of the combination of Hotspot and Random Value (LDS-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	1180990.2	256.8	1182368.1	401.86	100 %
sysmem_write_trans.	1650.4	135.79	1494.8	140.32	70 %

Table 6.31: The result of the combination of SRAD and Zero Value (LDS-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	361709.6	210.05	360463	228.97	100 %
local_store_trans.	4681202.5	37885.88	4713606.1	14350.27	70 %
stall_inst_fetch	2.00 %	0.02 %	2.09 %	0.02 %	100 %

test resulted in a probability of less than $0.1\,\%$. So in summary, Welch's t-tests show the statistical significance of the monitored symptoms for all benchmarks in the CPU experiments.

For the symtpoms observed during the GPU experiments, I again computed Welch's t-test using a feature of Microsoft Excel [125]. Excel's t-test function returns the probability that the means of the two input data sets belong to the same population. A value variation of a metric was accepted as a symptom if the probability returned by the t-test function was below $0.5\,\%$. Exemplary, the results of three t-test computations are shown in this work.

Table 6.36 shows the results for the t-test of the combination of SRAD with the instruction group GPR and the double-bit-flip fault model. The result heavily implies that both changes in the observed mean are not the consequence of differing values from the same population. So, it can be assumed that the symptoms reflect the occurrence of a fault during execution.

The same is true for the results computed for NN and the injection of the Random Value fault model into FADD-FMUL-OP instructions (s. Table 6.37). For all symptoms observed, the probability that the mean of the fault-free runs and the mean of the runs

Table 6.32: The result of the combination of Hotspot and Zero Value (LDS-OP)

Symptom	w/o faults		w faults		
	∅	s	∅	s	occ. r.
cf_issued	1237538.8	672.98	1238319.6	323.68	90 %
dram_write_trans.	7713710.2	1680.25	7716905.1	2139.32	70 %
l2_l1_read_trans.	2842370.4	873.18	2843818.8	1392.31	50 %

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICM	17.95	-11.66	0.001	-3.922	$8.26 \cdot 10^{-10}$
PAPI L2 ICH	10.94	9.04	0.001	4.437	$2.09 \cdot 10^{-6}$
PAPI L3 ICA	17.84	-23.16	0.001	-3.922	$1 \cdot 10^{-14}$
PAPI TLB DM	12.93	-5.04	0.001	-4.221	$2.31 \cdot 10^{-4}$
PAPI L2 DCA	9.00	9.50	0.001	4.781	$5.47 \cdot 10^{-6}$

Table 6.33: Welch's t-test results for the combination of SRAD and dial

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICM	17.92	-4.37	0.001	-3.922	$3.71 \cdot 10^{-4}$
PAPI L2 ICM	16.13	-3.16	0.01	-2.898	0.006

Table 6.34: Welch's t-test results for the combination of Hotspot3D and leak

with faults belong to the same population is at maximum $0.41\,\%$.

The t-test function results 6.38 for the symptoms monitored during the zero value fault-injection into the instruction group LDS-OP of Hotspot manifest the impression that the faults result in symptoms during execution.

Results Discussion

In this section, I evaluated the concept of symptom-based fault detection both on CPUs and GPUs. In particular, its capability to detect different faults and its utility to distinguish faults is investigated. The evaluation was done with the background that symptom-based fault detection shall be used in this thesis to compute an online fault rate as basis for a heuristic reliability metric.

For the evaluation, software-based fault injection in the form of the fault injection tools FINJ and SASSIFI, and intra-process communication to alter certain variables are used.

FINJ, the tool utilized for the CPU experiments, was used to create interferences that are deployed to mimic anomalies in real-life systems by stressing single components,

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICA	10.38	-69.1	0.001	-4.437	0
PAPI L3 ICA	16.79	-11	0.001	-3.965	$4.3 \cdot 10^{-9}$
PAPI PRF DM	14.35	-14.59.16	0.001	-4.073	$5.24 \cdot 10^{-10}$
STALLS TOTAL	11.37	-3.78	0.01	-3.012	0.0029

Table 6.35: Welch's t-test results for the combination of mMult and copy

Symptom	p
cf_issued	$7.49 \cdot 10^{-8}$
stall_inst_fetch	$9.62 \cdot 10^{-7}$

Table 6.36: Welch's t-test results for the combination of SRAD, the instruction group GPR, and the fault model Zero Value

Symptom	p
atomic_trans.	0.0047
cf_issued	$3.31 \cdot 10^{-}3$
inst_issued	0.031
issue_slots	0.032
l2_l1_write_trans.	0.041
local_load_trans.	0.0067

Table 6.37: Welch's t-test results for the combination of NN, the instruction group FADD-FMUL-OP, and the fault model Random Value

emulating interference or malfunction in that component. Additionally, a loop index alteration and iteration number reduction were conducted on the CPU via intra-process communication. In general, minimally two symptoms were found for every benchmark fault combination in the CPU experiments, which match the expected behavior change. All symptoms were then confirmed with Welch's t-test. In the worst case, there is at least one symptom with an occurrence ratio of $80\,\%$ and for most cases at least one symptom with a ratio of $100\,\%$. So, it is fair to say that it was possible to detect all faults in every benchmark on the CPU using symptom-based fault detection.

Faults that alter the number of executed instructions are easily detectable as these counters are very precise and therefore identical symptoms for all considered benchmarks altering loops were detected. A distinction from the other fault classes is also easy, as they do not alter the number of instructions.

Considering each benchmark on their own, the different instances of the interference

Symptom	p
cf_issued	0.0079
dram_write_trans.	0.0026
l2_l1_read_trans.	0.018

Table 6.38: Welch's t-test results for the combination of Hotspot, the instruction group LDS-OP, and the fault model Zero Value

classes (memory-bound and ALU-bound interferences) on the CPU had very similar behavior. E.g., SRAD showed significant variations in total L3 cache misses, the number of cycles needed, and TLB DMs for all three memory bound benchmarks. However, there was no single set of symptoms that was relevant for every instance of an interference class over all benchmarks. Only significant variations of instruction cache accesses and misses were visible for each instance of the interference classes over all benchmarks. A possible distinction could be the degree to which the values increase. ALU-bound interferences create larger increases compared to memory-bound ones. Additionally, in most cases, the memory-bound interferences created more variations in data related counters. Differentiating between the interferences and the loop index manipulation is hardly possible as these fault classes mostly affect cache structures and therefore create very similar symptoms. It is also noteworthy, that the probability of all symptoms occurring simultaneously is quite low as not all symptoms could be observed in every iteration.

In conclusion, this means that symptom-based fault detection is very useful to detect faulty application behavior and coarse-grained conclusions about the causing fault may be possible, but finer distinctions need additional tool support.

For the GPU experiments, SASSIFI was used to inject different faults into registers during execution by assembly-level instrumentation. In total, three fault models were injected in three instruction groups of three Rodinia benchmarks. Symptoms could be detected in every benchmark and the monitored symptoms again match the expectations belonging to the injected faults. However, in contrast to the CPU experiments, only one symptom was observable for Hotspot in combination with the instruction groups GPR and FADD-FMUL-OP and the symptom was not observable in every iteration. Still, the results confirm the overall impression that symptom-based fault detection is able do detect a wide range of faults. Again, I confirmed my measurements using Welch's t-test proving their statistical significance.

Distinguishing faults, however, remains nearly impossible on the GPU as well. Several symptoms are visible for multiple benchmarks over different instruction groups and fault models and again, not all symptoms are observable in all iterations. As this severly reduces the probability that all symptoms occur simultaneously, the ability to distinguish faults by comparing symptoms decreases even more. A potential problem may be only injecting faults into a single thread during execution as this limits the impact a fault can cause. Further experiments with the injection of faults in multiple threads in future work could lead to additional insights.

6.4 Reducing Profiling Overhead

The creation of a performance behavior database for task costs prediction and especially for scalability characterization requires a huge amount of profiling runs. Profiling runs create additional overhead, particularly if non-optimal task mappings are profiled.

To reduce the amount of profiling runs needed, I introduced two different approaches. The first approach employs scaling checks and interpolation to avoid unnecessary profiling runs and to utilize already stored information to predict unknown problem sizes or thread numbers. The objective of the second approach is to avoid profiling altogether by deploying machine learning algorithms to predict task execution times.

6.4.1 Interpolation & Scaling Checks

To reduce the profiling overhead, scaling checks and interpolation have been added to HALadapt's monitoring component in this thesis. The scaling characterization of OpenMP kernels requires profiling runs with varying numbers of CPU cores. A full exhaustion of a multicore CPU with n processing cores would demand n profiling runs. The first mechanism decreases this number by skipping thread numbers and using inter- and extrapolation to predict the execution time for the skipped profiling runs. Inter- or extrapolation are only allowed if the distance between the measurement points, and in case of extrapolation additionally the distance between the extrapolation candidate and the measurement points, is smaller or equal than a selectable threshold. For candidate thread numbers smaller or equal than ten, the distance threshold is set to four for interpolation and two for extrapolation. For greater thread numbers, I double the thresholds. If appropriate measurements have been found, the actual inter- or extrapolation computation depends on the belonging sensor. The execution time is interpolated/extrapolated by first computing the thread ratio efficiency of the measurements $E(n) = \frac{S(n)}{n}$, where n is the thread ratio and $S(n)$ the speed up. Depending on if the closest measurement is smaller or greater than the candidate's thread count, the execution time is divided by or multiplied with the product of $E(n)$ and the thread ratio of the measurement and the candidate. For the other sensors, we use linear inter- and extrapolation.

Additional runs can be avoided by stopping profiling runs iff the execution time scalability threshold of an OpenMP task is reached. A scaling check determines if a kernel is no longer scaling with additional threads, in this case no further measurements are conducted. To perform the scaling check, the first step is searching two measurements previously stored in the history database. Then, the thread ratio efficiency $E(n)$ for these two measurements is computed. If the efficiency is below a definable threshold, the kernel is determined to be not scaling with additional threads. In this case, the closest available measurement is returned as prediction for the execution time sensor. For the other sensors, a linear extrapolation is performed to calculate a prediction.

To further reduce the profiling overhead, creating task execution time prediction models based on static code analysis metrics [14, 88] is studied and evaluated. The next section explains static code analysis and the methods that are used to create the prediction models in detail.

6.4.2 Predicting Task Execution Times

In this thesis, a methodology based on a source code analysis using Clang and LLVM, and machine learning techniques to predict the fastest processor for a given OpenCL task by classification, and to predict task execution times is developed. The focus is set on OpenCL tasks as they can be executed on a wide range of different processing units and thereby are extremely well suited for today's systems featuring heterogeneous architectures. The following sections present the necessary fundamentals of source code analysis and machine learning necessary for this methodology. Additionally, implementation methods and details for the creation of the prediction models are elucidated. Finally, the evaluation of my prediction approach is presented.

Source Code Analysis

The methodology uses source code analysis to extract metrics from the source code of OpenCL programs. Source code analysis can be fielded into static and dynamic techniques. Dynamic methods try to extract characteristics of a given source code during runtime by instrumenting the code, which can influence, prolong, and alter the actual execution. Static code analysis on the other hand limits the observation to the source code itself. The analysis can happen before or after the compilation of the program. Also, a possible intermediate representation could be the target of the analysis. But since there is no information about the actual execution of the program, the analysis can deliver, especially for loops and branches, only an abstract view on the execution. Another disadvantage is, that optimizations by compilers cannot, or in case of the analysis of an intermediate representation can only partially, be taken into account. Because the goal is to reduce the necessary profiling overhead at the start of an execution cycle, only a static code analysis is used, complemented by some dynamic metrics, which are known before the actual execution through OpenCL's dynamic compilation approach.

The source code analysis is implemented by using the Clang tooling from the LLVM compiler suite [126]. With Clang tooling the built up abstract syntax tree can be traversed. The different source code constructs can then be observed by visitor functions which get called if a specific construct is found in the source code. In this way, metrics like binary operations or memory accesses can be counted. The binary operations are counted for each data type separately. If a construct occurs within a loop or a branch the metrics are multiplied or divided by a factor which is a parameter of the analysis and can be set before the analysis starts.

Beside these counted occurrences of different kinds of operations, two complexity metrics are also extracted. The used complexity metrics are the **NPath Complexity** from [127] and the **Cyclomatic Complexity** from [128]. The implementation calculating these complexity metrics is taken from [129].

Since a static code analysis can only provide an abstract view, the analysis is im-

proved by a branch prediction and a loop detection. The prediction is based upon variables whose values are known by, e.g., visited defines, given kernel arguments, or being the return value of functions returning a fixed value like work group functions. By calculating the visited operations, which use known variables, estimations of the probability if a branch is taken or not and the number of loop iterations can be made. With these predictions, the multiplier for the metrics occurring within loops or branches is adjusted.

Because not every loop or branch condition depends on variables whose values are fixed for all work items, the prediction is enhanced by the value range propagation from [130]. With the value range propagation, for each variable a value range and a probability is saved. Since there is no guarantee of being correct while calculating with these *probabilistic variables*, the range distribution of the values is always considered as *one*. With these *probabilistic variables*, predictions could be made for more loops and branches.

Following is an overview of the extracted metrics:

Number of binary operations		
• char • short • int	• long • half • float	• double • bool
Memory accesses		
• number of global read accesses	• read amount from global memory	• number of global write accesses
• written amount to global memory	• number of local read accesses	• read amount from local memory
• number of local write accesses	• written amount to local memory	
• proportion of global memory accesses to the number of total binary operations		
Loops and branches		
• number of loops	• number of branches	• average depth of nested loops
• average depth of nested branches	• max. depth of nested loops	• max. depth of nested branches
• number of nested loops	• number of nested branches	
Number of called OpenCL functions		
• atomic and asynchronous	• mathemtical	• other
• number of kernel arguments	• number of buffer kernel arguments	• number of array accesses
• synchronization points	• NPath Complexity	• Cyclomatic Complexity
• declarations of variables	• problem sizes	

Table 6.39: The extracted source code metrics utilized to create the prediction models

Prediction Model Creation

To create the classification and prediction models, machine learning algorithms are employed. Machine learning can be split into supervised, unsupervised, and reinforcement learning [131]. Supervised machine learning tries to infer a function from labeled training data [132]. Algorithms generate this inferred function by analyzing the training data. This function can then be used to map new examples. The training data consists of pairs of feature vectors and a label. These pairs are also known as training examples. The label is predicted by the inferred function for future examples and can be a category or a real number. In the first case, the problem is called classification and in the latter regression. The inferred function is called machine learning model.

To predict the fastest of a set of processors, a category is used as label. Therefore, classification algorithms of supervised machine learning are utilized to solve this

problem. The task execution time prediction uses past measurements as labels. So, for this problem regression algorithms of supervised machine learning are employed. In summary, the following algorithms are used:

- ***k*-Nearest Neighbor** tries to find the *k* nearest points of an example to be predicted in the space of the feature vectors by computing the distance to all points via a distance metric. The most frequently represented label among the *k* nearest points is then predicted [133]. It can be utilized both for classification and regression.

- **Decision Trees** are machine learning models which make predictions by learning simple decision rules inferred from the training data. Starting at the root of a tree, in each step the training examples are split into two parts by a threshold of a single feature. The leafs of the tree determine the prediction for a new training example. **Random Forests** are a combination of multiple Decision Trees and were first mentioned in [134]. To make a prediction, each Decision Tree predicts a value. The value which gets predicted the most, is the prediction result.

- **Support Vector Machines** (SVMs) try to divide the training examples in the space of the feature vectors by a hyperplane. The optimal hyperplane is determined by maximizing the distance of a to be selected distance metric to the feature vectors [133].

- **Multilayer Perceptron** is one of the simpler neural networks [135]. It consists of an input, an output, and at least one hidden layer. These layers are composed of artificial neurons, called perceptrons. Perceptrons map an input vector x to an output $f(x)$ by weighing each input vector element and then passing the sum through a non-linear function [136]. Training of a multilayer perceptron is done by backpropagation [137].

The machine learning parts are implemented with the Python programming language making usage of the scikit-learn [138] library.

To train the machine learning models, training data has to be created. As training data, the execution times of about 270 OpenCL kernels were measured and the kernels afterwards analyzed. The OpenCL kernels were taken from the following software development kits and benchmarks:

- AMD APP SDK [139]
- Intel OpenCL Samples [140]
- Hetero-Mark [141]
- Parboil Benchmark [142]
- PolyBench/GPU [143, 144]
- Rodinia Benchmark Suite [123]
- SHOC Benchmark Suite [145]
- Hydro2de [146]

By using HALadapt's OpenCL wrapper, the execution time of the OpenCL kernels can be measured. The OpenCL wrapper acts as an OpenCL device and passes the OpenCL

function calls to a real OpenCL device. Thus, the wrapper has access to detailed information and parameters of the OpenCL kernel and besides measuring the execution times, the values of committed kernel parameters, problem sizes, and build parameters can also be logged. This only creates overhead for the host but does not influence the execution on the device itself.

To generate the training examples, the OpenCL kernels were executed with varying problem sizes on all of the ten processors. With each problem size, the kernel was run at least ten times. After measuring the execution times, the source code of each OpenCL kernel was analyzed by the developed code analysis (s. 6.4.2). The extracted code metrics were then combined with the problem size to a feature vector. As label, either the device number of the OpenCL device that executed the related kernel the fastest or the measured kernel execution time was used. So, each feature vector contains the collected code metrics of the associated OpenCL kernel and is labeled either with the device number of the OpenCL device that executed the kernel the fastest in the considered evaluation scenario or the measured kernel execution time. This means the labels can change over different evaluation scenarios for the classification, e.g., the set of considered processors changes, and the model has to predict the correct label for a given OpenCL kernel.

To calculate an average over the multiple executions of each kernel, which results in many slightly different code metric values, and also to minimize a bias towards a specific kernel, vectors with *similar* code metric values are combined. Two vectors are defined as being *similar*, if all features at most differ by *five* percent. Afterwards, the vectors are normalized.

Out of the resulting vectors two sets are created. 75 percent are forming the training set, the other 25 percent the validation set. The validation set is exclusively used to evaluate the final machine learning models and is not employed to train the models, so the evaluation can be done with apriori unknown data.

The training set is then utilized to train the machine learning models. To determine the optimal parameters for the model generating algorithms, a grid search trains multiple models. Cross Validation is employed to determine the performance because thereby it is not necessary to use a subset of the training set exclusively for evaluation. In Cross Validation, the training examples are split in several sets [147]. A model is generated for each set, which is used as validation set. The remaining sets are used as training examples to generate the model. To estimate the performance, each model predicts values for the examples of the validation set. The prediction performance for the whole set is then determined by taking the average. This allows to not further divide the training set into a training set and an evaluation set that evaluates the parameters. The best performing model is taken for further experiments and is then, as mentioned previously, validated with the validation set.

The impact of the different features is determined by the ANOVA F-Test [148] (6.1), which is defined as follows for a set of training vectors x_k with n_+ positive and n_- negative instances:

$$F(i) = \frac{\left(\bar{x}_i^{(+)} - \bar{x}_i\right)^2 + \left(\bar{x}_i^{(-)} - \bar{x}_i\right)^2}{\frac{1}{n_+-1}\sum_{k=1}^{n_+}\left(x_{k,i}^{(+)} - \bar{x}_i^{(+)}\right)^2 + \frac{1}{n_--1}\sum_{k=1}^{n_-}\left(x_{k,i}^{(-)} - \bar{x}_i^{(-)}\right)^2}, \tag{6.1}$$

where $\bar{x}_i, \bar{x}_i^{(+)}, \bar{x}_i^{(-)}$ are the average of the $i-$th feature of the whole, positive and negative data sets, respectively; $x_{k,i}^{(+)}$ is the $i-$th feature of the $k-$th positive instance, and $x_{k,i}^{(-)}$ is the i-th feature of the $k-$th negative instance. The calculated impact is then validated by training models with a reduced feature set. The number of utilized features is halved until only one feature is used for training. Every time, the lower scored features are removed.

Related Work for Execution Time Prediction

In [149] a machine learning model is used to predict the expectable speed up for the portation of CPU code to a GPU and the best device for a given OpenCL kernel out of a multi-core CPU und two GPUs. Baldini et al. collect dynamic code features by binary instrumentation using Ping and use these and the speed up of a 12-threaded OpenMP implementation as features for the prediction. Binary instrumentation usually alters the application execution and the additional instructions create overhead during runtime. As Baldini et al. state in their paper, the utilization of performance counters could be better suited for runtime contexts like scheduling. This thesis evaluates bigger sets and with the Xeon Phi an additional accelerator. Additionally, the prediction of task execution times is also evaluated.

In [150] the goal is to support the scheduling of OpenCL kernels in a system consisting of a CPU and a GPU. This is achieved by predicting speed up categories for OpenCL source code to differentiate between an execution on a GPU or a CPU by classifying a kernel to low or high speed up on the GPU. Beside some static code metrics, the approach also uses the input and output sizes and thread numbers.

The goal of [151] is to automatically select the fastest OpenCL device for the usage of Java's parallel stream API. This is done by implementing a classification directly in the Java JIT compiler. Contrary to this thesis, the predictors have to differentiate only two processors making the prediction easier.

Wu et al. [152] employ machine learning to predict the performance of a GPU kernel on different target architectures belonging to the training set. The prediction is based on previous executions of the kernel on a base hardware configuration and collected performance counter values. This is done by a classifier which maps the kernel to clusters representing different scaling behaviors. In contrast to the work presented in this thesis,

Wu et al. are only focused on the performance of kernels on GPUs and additionally need an execution of every kernel to predict its performance.

Amarís et al. [153] predict the execution time of applications using vector operations on NVIDIA GPUs via different machine learning algorithms. The models utilize dynamic code metrics and architecture characteristics like the number of cores and the maximum GPU clock rate as features. As the work of Wu et al., Amarís et al. also only focus on GPUs and do not consider CPUs or other accelerators. Additionally, they only consider a very specific set of applications.

In [154], Hoste et al. proposed determining the similarity of a program to a known benchmark database to predict the program's performance speedups on different machines, specifically which machine provides the biggest speedup. Similarity is based on a set of microarchitecture-independent characteristics like the instruction mix, instruction-level parallelism, register characteristics, and branch behavior that are collected for the database and the new program. The values of the database benchmarks are stored in a matrix, which is then transformed by either normalization, principal component analysis, or genetic algorithm to a data transformation matrix that is used to compute the performance prediction. The performance prediction is computed by taking a weighted average over the performance numbers of the benchmarks in the neighborhood of the program. The weights are proportional to the inverse of the euclidean distance between the program and the benchmarks. This method requires similar benchmarks to be able to make a prediction for a new program and compared to my approach only predicts relative performance.

In the two following works, an optimal task partitioning for the simultaneous execution of OpenCL on multiple processors is predicted. In both works, mainly static code metrics are used as features for the machine learning algorithms, complemented by some runtime features like the number of threads or transfer sizes between host and device memory. In [155], the prediction distinguishes between an execution on either the CPU or the GPU or a mixed execution. The prediction of the optimal distribution on the multiple processors is then done in a second step. Kofler et al. [156] use Support Vector Machines as well as Artificial Neural Networks to find the optimal task partitioning in a single step.

Evaluation of the Prediction Models

The prediction model experiments include a variety of different processors on which the kernels were executed. On the side of the CPUs there are the Intel I7-6700k, the Intel i7-5820k, and a system with two Intel Xeon E5-2670v2 processors. NVIDIA GPUs are covered by the NVIDIA GTX 980 Ti and the NVIDIA GTX 650 Ti Boost. AMD GPUs are represented by the Radeon HD 7970, Radeon HD 7750, and the Radeon RX 470. Beside these models the Intel Xeon Phi 7120 and the Intel HD Graphics 530 are included.

The OpenCL kernels were executed on an Arch Linux with the Linux Kernel in version 4.7. For the AMD devices the AMD Catalyst driver was used in version 15.9, for the NVIDIA devices the GeForce 367.27 driver. The Intel devices were driven by three different drivers. The Intel CPUs were used with the Intel OpenCL Runtime 16.1.1 and the Intel Xeon Phi accelerator with Version 14.2. On the Intel GPU, OpenCL was executed with the Intel OpenCL driver 3.0.

As evaluation metric for the classification experiments *accuracy* (6.2) is used. Accuracy is defined as the ratio of correct to all predictions.

$$accuracy = \frac{correct\ predictions}{total\ number\ of\ predictions} \tag{6.2}$$

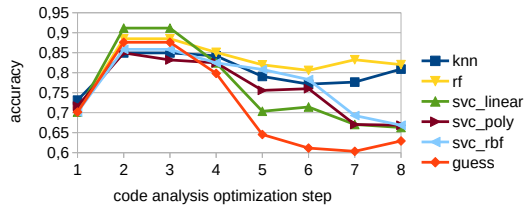
The regression experiments are evaluated with the *symmetric Mean-Absolute-Percentage-Error* (*sMAPE*) [157] (6.3). sMape is defined as follows:

$$sMAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{\|y_i - f_i\|}{(\|y_i\| + \|f_i\|)/2}, \tag{6.3}$$

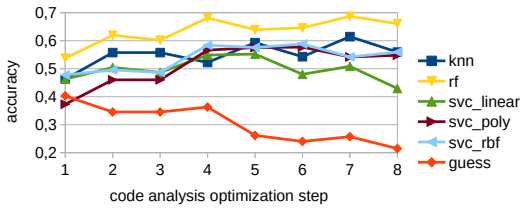
where y_i is the correct value and f_i the prediction value.

The OpenCL source code was analyzed with eight different settings to enable the evaluation of these optimization steps. At first, the branch prediction and loop detection were disabled, and the problem size and values of kernel arguments not taken into account (**step 1**). All metric values in loops and branches were multiplied with *one*. In a next step, the problem sizes were taken into account and metric values in branches were divided by *two* (**step 2**). After this, the branch prediction was enabled without probabilistic variables taken into account (**step 3**). In the following step, the values of kernel arguments were considered (**step 4**) and afterwards the branch prediction was enabled with probabilistic variables (s. Sec. 6.4.2) (**step 5**). In the two following steps, the multiplier in loops was set to *two* (**step 6**) and after this the loop detection was enabled (**step 7**). In a last step, the multiplier in loops was set to *16* (**step 8**). It should be noted that these optimization steps are independent of each other and can be used in various different combinations and in any order.

The Fastest Processor of a Set In this experiment, the task is to select the fastest processor out of a set of heterogeneous processors for a given, but to the machine learning model unknown, OpenCL kernel. For this experiment, two sets were generated. The first set consists of every available processor. For the second set, the three fastest and so most dominating processors, the AMD HD 7970, the NVIDIA GTX 650 Ti Boost and the NVIDIA GTX 980 Ti, were taken out to make the set more even and thereby the classification harder. For both sets different machine learning models were trained. In Fig. 6.5a and Fig. 6.5b, the prediction performance is shown for five different models generated by random forests (rf), *k*-Nearest Neighbour (knn), and Support Vector Machines (SVMs).



(a) Set 1, all processors.



(b) Set 2, all processors except AMD HD 7970, NVIDIA GTX 650 Ti and NVIDIA GTX 980 Ti.

Figure 6.5: The accuracy for the trained models with different settings of the code analysis (s. 6.4.2), selecting the optimal processor of a set.

Three different kernels are used for SVMs, namely a linear (svc_linear), a polynomial (svc_poly), and a radial basis function (svc_rbf). These five models were trained for the eight different settings of the code analysis (s. Sec. 6.4.2).

A guessed (guess) prediction acts as a baseline, which always predicts the processor, which is the fastest in most cases as being the fastest for every OpenCL kernel. This results in a much higher baseline for validation sets being dominated by one device than a totally random guess. This means that the processor that executes most OpenCL kernels the fastest is always chosen as prediction.

As the first set is strongly dominated by the NVIDIA GTX 980 Ti, the machine learning models could hardly deliver a better prediction as a guessed result. This is especially the case for the first settings of the code analysis, as can be seen in Fig. 6.5a. By activating the branch prediction with probabilistic variables not only the executions are represented better by the training examples, but the models also deliver better results. The best predictions are made by the random forest model with the optimization step seven. This model achieves an accuracy of 0.83 compared to 0.6 of the guessed result.

The second set is clearly more balanced (s. Fig. 6.5b), as one device is only the fastest for about 25 percent of the kernels. Although multiple devices are represented by a similar amount of training examples, the random forest models can deliver an accuracy of up to 0.69 in contrast to a guessed result with an accuracy of 0.25. This was accomplished by activating the branch prediction with probabilistic variables and the loop detection.

For these experiments, the best results are achieved with the models being trained with all extracted features. Despite this, adequate results could be delivered with a reduced number of features. With a fourth of the features the models are still able to distinguish themselves from a guessed prediction. Especially for the second set, the achieved accuracy of 0.62 was only slightly lower.

	Set 1	Set 2
#1	array accesses	global memory writes
#2	binary operations	global written size
#3	binary operations (int)	global memory reads
#4	global memory reads	array accesses
#5	global read size	binary operations
#6	number of loops	binary operations (int)
#7	number of nested loops	global read size
#8	binary operations (double)	Cyclomatic Complexity
#9	binary operations (float)	binary operations (float)

Table 6.40: The most important features for the experiments with sets of processors.

Table 6.40 shows the features that have the most impact. Among the most important ones are numbers of binary operations, array accesses, and numbers describing the global memory access. For the second set the *Cyclomatic Complexity* also has a relatively high impact.

Predicting OpenCL Kernel Execution Times The goal of this experiment is predicting the exact kernel execution time for a given, however unknown to the machine learning model, OpenCL kernel on a specific, known processor. In this experiment, for each processor a specific machine learning model was created with each algorithm. Next to the five algorithms used in the classification experiment 6.4.2, a multilayer perceptron with one hidden layer and 100 units was used to generate a prediction model. Again, all models were trained for the eight settings of the code analysis (s. Sec. 6.4.2). A guessed prediction (guess) that always predicts the average over all training examples acts as a baseline.

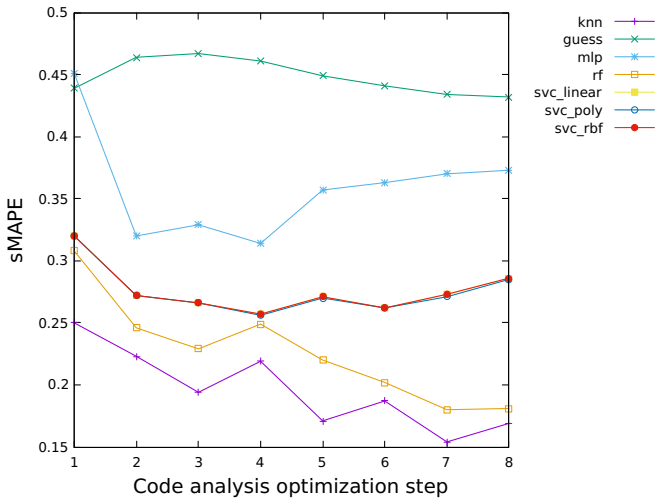


Figure 6.6: sMAPE results for prediction of the kernel execution times on the NVIDIA GTX 980 Ti with different settings of the code analysis (s. 6.4.2)

Figures 6.6, 6.7, and 6.8 present the results for the prediction models for kernel executions on the NVIDIA GTX 980 Ti, Intel's i7-6700k CPU, and AMD's Radeon RX 470 GPU, respectively. The overall best results are achieved by the model generated by *k*-Nearest Neighbor with the second best results stemming from the random forest models over all three processing units.

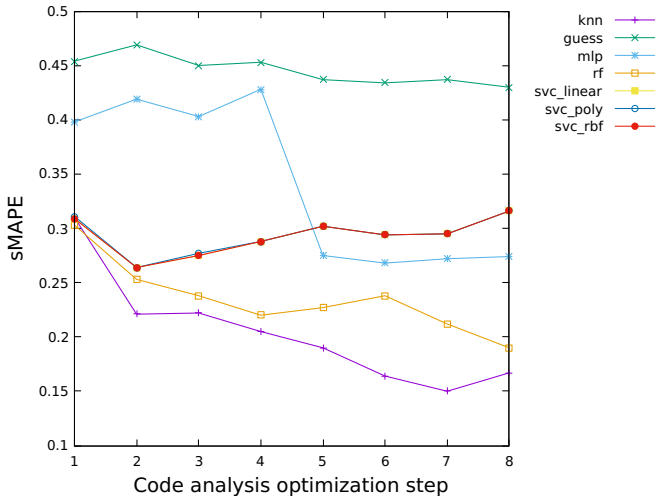


Figure 6.7: sMAPE results for prediction of the kernel execution times on the Intel i7-6700k CPU with different settings of the code analysis (s. 6.4.2)

It is noteworthy, that there is almost no distinction between the different SVM kernel functions and the SVM models could not benefit from the code analysis optimization steps for all three processors. For almost all optimization steps, the models created by the SVM algorithms and the multilayer perceptron models achieve distinctively worse results than the rf and knn models. In general, all models outperformed the guess baseline.

However, even the best average results still deviate quite heavily from a perfect prediction. This is emphasized by taking a look at the individual predictions and their percental deviation from the actual execution time. Figure 6.9 shows the percental errors for all predictions made for NVIDIA's GTX 980 Ti by the random forest and *k*-Nearest Neighbor models with optimization step 8. A desirable outcome would be an accumulation of predictions in the range $[-0.1, 0.1]$, i.e. a prediction error of at most $10\,\%$. The results, though, comprise a large number of under- and overpredictions including predictions of over $5000\,\%$ and under $10\,\%$ of the original execution time. Still, there are about 50 kernels for which the prediction falls in the desired range.

Similar patterns are visible for the individual predictions of the execution times on the Intel i7-6700k CPU. The histogram of the percental errors for the *k*-Nearest Neighbor and random forest models with optimization step 8 can be seen in Fig. 6.10. Here, about 40 predictions fall in the desired range of $[-0.1, 0.1]$ and again, there are a con-

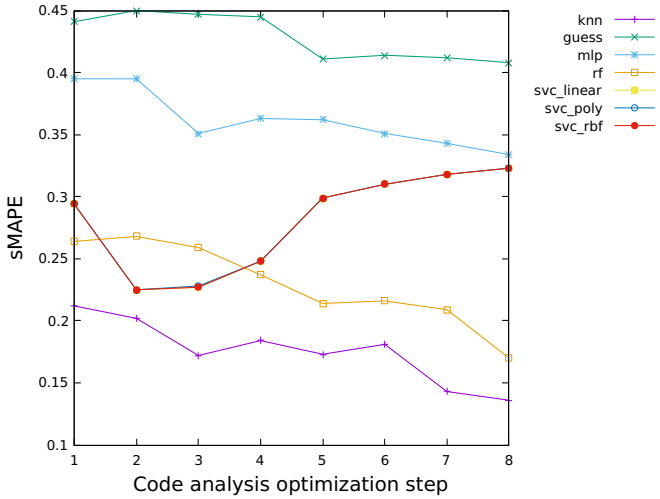


Figure 6.8: sMAPE results for prediction of the kernel execution times on the AMD Radeon RX 470 with different settings of the code analysis (s. 6.4.2)

siderable amount of heavy over- and underpredictions. In summary, the results show that the generated prediction models are not able to reliably predict the execution time of all kernels. However, the prediction in general returns useful results for quite a large amount of OpenCL kernels. In future work, these kernels should be analyzed for similar characteristics and potentially grouped into classes. In the best case, this allows to differentiate the kernel whose execution time can be predicted by a model from the other kernels. Thereby, it would be possible to create a test that beforehand decides if a kernel execution can reliably be predicted.

Results Discussion The first part of the prediction model evaluation focused on selecting the fastest processing unit within a set for different OpenCL kernels. For the first experiment, all eleven processors were considered. As the GPUs AMD HD 7970, NVIDIA GTX 650 Ti, and NVIDIA GTX 980 Ti dominated the set, i.e. they execute most kernels the fastest, they were excluded from the second set. In total, five prediction models were trained for each experiment and then compared to a guessed baseline that always predicts the fastest processor of the set. Both experiments showed the best results were obtained by the models created by the Random Forest algorithm. Especially for the harder set, the random forest model could still achieve an accuracy of around $70\,\%$ in contrast to the $25\,\%$ of the baseline. So, it is fair to say that in most cases it is possible to reliably predict the fastest out of a set of processing units for an unknown OpenCL kernel.

In the second experiment, the goal was not to select the fastest processor but to predict the actual execution time of an OpenCL kernel on a specific, known processing unit.

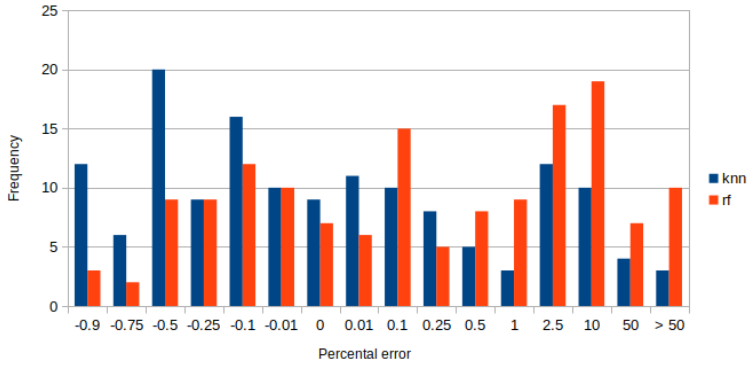


Figure 6.9: Percental error of the predictions by knn and rf for the execution times on NVIDIA's GTX 980 Ti

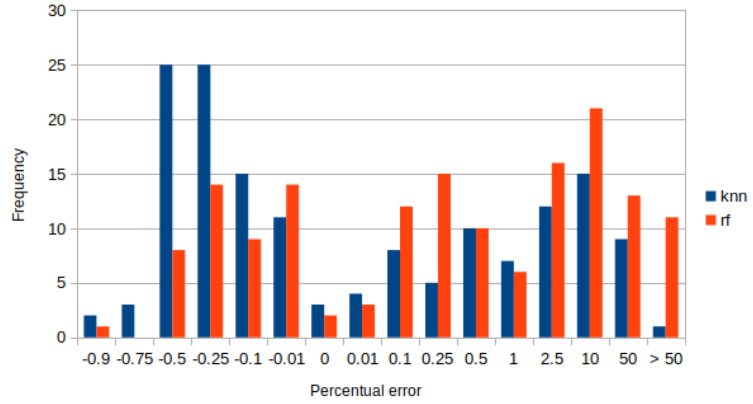


Figure 6.10: Percental error of the predictions by knn and rf for the execution times on Intel's i7-6700k CPU

A prediction model created by a multilayer perceptron was added for these experiments. As a comparison baseline, a guess prediction that always guesses the average execution time over all training examples was used. The six prediction models were evaluated by computing the sMAPE metric, which is defined as:

$$sMAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{\|y_i - f_i\|}{(\|y_i\| + \|f_i\|)/2}, \tag{6.4}$$

where y_i is the actual execution time and f_i the predicted time. The results showed a great discrepancy in the capability of the specific models. Clearly, the best results could be obtained using the *k*-Nearest Neighbor (knn) models with the random forest (rf) coming in second.

However, as the sMAPE values computed were not as close to zero as needed, I took a closer look at the prediction histograms for both the knn and rf models. The histograms showed that there are kernels for which the prediction works well resulting in predictions with a maximum deviation of $10\,\%$. However, they also showed that for a number of kernels both models produced huge deviations, both too small and too big. In summary, this means that the models are generally not able to reliably predict the execution time of an OpenCL kernel on a known processor. Still, the results contain potential for future work. If the kernels, whose execution time can be predicted reliably, possess similar characteristics that allows to separate them from the other kernels it would be possible to differentiate for which kernels a prediction is possible before doing the actual prediction. In future work, it therefore is necessary to analyze the kernels for characteristics and then confirm the classification with new examples and predictions.

6.5 Summary and Conclusion

The focus of this chapter is capturing the state of a system by monitoring its execution. HALadapt offers a mechanism that takes on this task and uses a database to store past measurements. In the course of this thesis, this mechanism is extended to support OpenMP thread variation and throttling, and CPU frequency scaling. Hereby, the number of cores used for the execution and the CPU frequency were added to the set of database keys that characterize each database entry.

The analysis of several project requirements and goals in Chapter 5 shows that system reliability becomes increasingly important. However, there is no hardware counter available that can just determine the reliability of the system or a system component. Though, in order to consider reliability in an adaptation process, some form of reliability measure is needed. Therefore, a heurstic realibility metric was created for this thesis and added to HALadapt's monitoring component. For the reliability metric, the idea is to use a lightweight fault detection mechanism and compute a fault rate based on past executions and the ratio of faulty to correct execution runs. As lightweight mechanism, symptom-based fault detection was chosen and its efficiency in detecting and distinguishing faults evaluated on both CPUs and GPUs. Symptom-based fault detection is based on the hypothesis that faults manifest themselves in behavioral changes during execution that can be detected by monitoring performance metrics. The experiments confirmed that it is possible to detect several distinctive faults using the concept of symptom-based fault detection. This enables the concept for the proposed heuristic reliability metric using symptom-based fault detection to create fault-histograms. However, it was not possible to draw conclusions about the causing fault by just looking at the occurring symptoms. A drawback of symptom-based fault detection and considering a wide spectrum of different performance metrics as potential symptoms is that only a limited amount of hardware counters are available in today's architectures and the number of counters is dependent

on the platform. Hence, multiple profiling runs may be necessary to collect all relevant performance data.

Creating an extensive database that stores the necessary information to make sophisticated decisions, e.g. in task-scheduling, requires a vast amount of profiling effort. Particularly, improving the efficiency of the number of threads used for OpenMP kernels requires a thorough exploration of the optimization space by profiling different execution runs. In order to reduce this profiling effort, this thesis presented mechanisms that enable HALadapt to skip profiling runs and predict necessary information. The first mechanism employs linear extra- and interpolation to heuristically predict the execution costs, i.e. the execution time or energy consumption, for unknown thread numbers. This allows to limit the profiling to certain thread numbers and use the collected data to predict the missing data. Additionally, a scaling check was added that recognizes if an OpenMP kernel does not scale with additional threads and then stops additional profiling runs for higher thread numbers. This approach, however, only works on the condition that the kernel exhibits a somewhat regular execution behavior pattern that can be detected by linear inter- or extrapolation.

To further reduce the necessary profiling executions, a prediction mechanism for the execution time of unknown kernels was developed and added to HALadapt. This allows HALadapt to gain information without the need for profiling runs. Hereby, OpenCL kernels are focused as they can be executed on a wide range of different processing units and by that specifically target heterogeneous systems. The mechanism uses static code analysis to collect kernel characteristics that are then utilized to train prediction models via machine learning algorithms. The results of the evaluation showed that the models were in general not able to reliably predict the execution time. However, the results also present potential for future work as further analysis showed that the prediction does work for a certain amount of kernels and that the models can reliably predict the best processing unit for an upcoming kernel. In order to verify if there exist certain characteristics that enable or disable a reliable prediction further evaluations and analysis is needed.

In total, this chapter extends the existing profiling mechanism of HALadapt and provides efforts to reduce the needed profiling overhead while collecting the necessary information to make sophisticated decisions. Thereby, this chapter lays the groundwork for the proactive mechanisms presented in the remainder of this thesis by providing the needed knowledge about the system itself and its environment. However, to achieve proactivity, additional knowledge about the future is needed. The following chapter introduces and discusses methods to predict future behavior.

CHAPTER

SEVEN

PREDICTING FUTURE SYSTEM STATES

The ability of a system to act proactively requires at least partial knowledge about the near future of the system and its environment. Therefore, the holistic approach of this thesis presented in Figure 4.1 in Section 4.2 includes the component Data Analysis & Prediction. The task of this component is to analyze the provided monitoring data and utilize the created history database to make predictions.

As this thesis targets task scheduling in dynamic heterogeneous systems, useful information about the future include the next tasks that have to be executed, the costs of the tasks to be executed, and changes in the system environment. The prediction of tasks' costs is extensively discussed in Section 6.4 in the context of profiling overhead reduction. There, a mechanism to predict task execution times for unknown OpenCL kernels based on static code analysis and machine learning is introduced. Additionally, extra- and interpolation for costs of unknown problem sizes and thread numbers for already profiled tasks is presented.

This chapter focuses on the prediction of future tasks to be executed that have not arrived in or are known by the system yet and offers mechanisms to predict newly arriving tasks and their starting point. The prediction is hereby based on the analysis of past execution patterns. In combination with the aforementioned tasks' costs prediction, this allows the system to proactively plan and consider side effects of future executions, and to avoid disadvantageous system states, e.g., overheating the system or a specific processing unit, or draining the energy budget before all tasks are executed.

This chapter is organized as follows. An introduction into the topic of this chapter and an overview of related work is presented in Section 7.1. Section 7.2 shortly introduces the necessary theoretical background to implement the prediction mechanisms of this chapter. Both prediction mechanisms are explained in Section 7.3. The results are presented in Section 7.4 and the chapter is summarized and discussed in Section 7.5.

7.1 Introduction & Related Work

A prerequisite for proactive behavior is some knowledge about the future as this allows a reaction before this future occurs and manifests itself, and opens up possibilities for a wide range of system optimizations. Therefore, existing implementations of proactive systems in the literature usually include one or more prediction mechanisms and a great variety of different mechanisms to predict the future in a wide area of fields of application has been studied and developed.

VanSyckel [41] implemented a task-based selection mechanism that dynamically selects a prediction method from a set of available methods based on a list of parameters like the data type and the parameter dimension of the prediction problem. The prediction set consists of an alignment approach, linear regression, a Markov model, a self-organizing map (SOM), and a state predictor based on branch predictors [158].

Engel et Etzion [40] extend the conceptual event-driven architecture to be able to act proactively. This includes prediction mechanisms, particularly several predictive agents. The first approach are rule-based predictive agents that detect an input event pattern and derive an output event with a belonging occurrence probability in a specific time interval. They also list limitations of rule-based predictive agents. As an arrival of a certain event usually changes the occurrence probability of other events, this also has to be expressed by rules. Additionally, input events can also be probabilistic, which increases the complexity of the probability computation of the derived output event. A model that can overcome these limitations is introduced in the form of Bayesian agents that consist of a directed acyclic graph (DAG) that models the probabilistic dependencies between the random variables of the underlying problem. To include temporal information in a Bayesian network, Dynamic Bayesian Networks [159] or Continous Bayesian Networks [160] are used. If there is not enough training data or explicit probability measures are not available, classifiers like decision trees (s. Section 6.4.2) are a suitable option according to Engel et Etzion.

A predicition of future GPS locations of a mobile device is implemented in [161]. Based on a trace of past locations, future locations are computed via interpolation. Thereby, the three interpolation methods, Newton's Divided Differences, Lagrange's Interpolation, and Cubic Bezier Splines, were evaluated. The best results were provided by Cubic Bezier Splines as the other two methods could not handle the increasing amount of input data.

Klös et al. [33] provide a lightweight online-learning approach that calculates future environment profiles based on past observations. The approach collects parameter values over several time steps and combines them into profiles. New observations are compared with the saved profiles within a set lookback window. If profiles match the current observation in this window, prediction profiles are created that contain the parameter values in a set lookahead window. In the case that several profiles match the current observation, the predictions are combined, e.g., by a weighted average.

State prediction also plays an important role in power management. Gao et al. [162] predict future voltage measurements using an auto-regressive (AR) model and past measurements.

An approach to predict future power system states based on Markov models (MM) and the Viterbi algorithm (VA) is presented in [163]. The approach creates a grid of feasible power system states based on historical data and then models the behavior of the power system states with a MM, explicitly computing the state transition probabilities. Then, VA is used to predict the most likely sequence of future states over a time period.

To forecast the future power demand of consumers, Hernandez et al. [164] deploy artificial neural networks (ANN) with weather variables and load values serving as input data. The input data is first classified using a SOM that clusters together demand patterns with similar features. The second stage of the approach selects online to which cluster new measurements belong. For each cluster, a specific ANN then predicts the load of the next 22 hours.

The focus of this thesis is to guide the future behavior of dynamic heterogeneous systems via task scheduling. Therefore, in constrast to the aforementioned publications, the goal of this chapter is to predict future tasks to be executed, that have not arrived in or are known by the system yet, by analyzing past execution patterns. Thereby, two different prediction scenarios are considered. The first scenario regards the execution pattern of the instances of a single task that is independent of other tasks in the system. Especially in embedded system, certain tasks are executed periodically or in the least with some frequencies and sporadically, i.e., with a minimum inter-arrival time between new instances. In the second scenario, the focus lies on the prediction of task set patterns or applications, i.e., tasks or task combinations that usually follow a specific task or task combination and therefore are dependent on past tasks. These scenarios require approaches that are lightweight and do not warrant a large amount of training data as they have to be performed online in dynamic systems and over a variety of different system configurations.

A research field that is similar to the goals of this chapter, is the prediction of future application phases. Most applications are comprised of sections of similar behavior, which are defined as phases with significant changes in behavior being defined as a phase change [165]. Several mechanisms have been studied and developed to predict future application phases.

In [166, 167], a run length encoding (RLE) Markov predictor, based on the concept Markov chains, is used to predict phases defined by clustering and the evaluation of system states defined by monitoring values of performance counters. A RLE Markov predictor is also utilized by Vandeputte et al. [168] and Sherwood et al. [169]. Vandeputte et al. use the predictor in an evaluation study of phase predictors comparing it to a Burst predictor and a Last Value predictor. Additionally, the predictor improvements "Confidence", a threshold of prediction verifications before a prediction gets accepted,

and "Conditional Update", a threshold of wrong predictions before the prediction table gets updated, are introduced. Another phase predictor evaluation was done by Hock et al. [170]. They implemented a Last Value predictor, a RLE Markov predictor, and a perceptron predictor.

Shen et al. [171, 172] solely use the LRU-stack distance to determine phase behavior in a program. The resulting phase hierarchy is expressed by using grammar compression. Thereby, phases are represented by regular expressions. In [171], a history-based predictor predicts upcoming phases. The predictor monitors the program execution and recognizes the current phase by matching the regular expression with a finite automaton. Based on the hierarchy, the next phase is then predicted. A different approach is presented in [172]. Here, only the exponents of the regular expression change for different inputs. An input component analysis is employed to predict the exponents.

Multi-level phase analysis and prediction is introduced in [173]. Fang et al. separate programs in coarse- and fine-grained phases, where a coarse-grained phase is composed of stably-distributed fine-grained intervals. The fact that coarse-grained phases can be identified by a short interval of fine-grained phases at the beginning is then used for the prediction mechanism. First, the fine-grained phases are predicted using a standard phase prediction mechanism. If enough fine-grained phases have been executed to identify the coarse-grained phase, the coarse-grained phase information is utilized to predict the remaining fine-grained phases.

Similarly to predicting the program phase behavior, Quan et al. [174] use a prediction mechanism to predict the next task in a sequence of tasks in the context of HW/SW partitioning. The mechanism detects periods in a task sequence sample and then uses this period to predict future tasks. If no period can be detected, Markov transition probabilities are heuristically determined and utilized to compute predictions. However, no temporal information is used or predicted.

Although phase behavior prediction and predicting future tasks and task patterns in this work share some similarities, there are some major differences. Per definition, there is only one phase running at all times whereas several task instances of different tasks may run in parallel in the fields of application in this thesis. Thus, interferences between several task execution patterns are possible and have to be considered in the creation of a prediction mechanism. Additionally, phases always directly succeed a predecessor phase. Task instances, however, may have arbitrarily long delays between each other, which further complicates the prediction process. This means a prediction mechanism for the purpose of this thesis also has to include temporal information of the task executions to correclty identify execution patterns.

In [175] workflows of HPC clusters are predicted as observations showed that the job submissions of the users show periodic behavior. For the prediction mechanism, the task sequence is interpreted as a time series. As the standard time series prediction algorithms ARIMA [176], GMDH [177], and SSA [178] did not provide reliable results, a

deep learning method was deployed. Therefore, the time series were first decomposed by a hierarchical clustering approach deploying the Mahalanobis distance [179] to reveal patterns of recurrent tasks. A neural network is then trained for each pattern with task and pattern features as input data and used to answer the question "will a certain task appear in a definite instant of time?".

In summary, approaches based on MMs and pattern matching extended with the necessary temporal information seem to be the best choice for this thesis as they are lightweight and do not need extensive training data, probability measures, or a long training phase. Particularly, as probability measures are often hard to compute for large systems, and extensive training data may not be available and long training phases not possible in dynamic systems, these features are of great importance for the approach of this thesis. However, if enough training data is present, the approach based on time series and a neural network presented in [175] also seems promising. The following section introduces some necessary background to implement an approach based on MM and pattern matching.

7.2 Theoretical Background

This section elucidates the necessary theoretical background of the prediction mechanism created in this chapter. First, Markov chains are introduced in Section 7.2.1. Markov chains are an important theoretical concept as they are based on the observation that a future state relies only on the current state. Predictors that are based on the Markov chain concept, called Markov predictors, are introduced in Section 7.2.2. Additionally, this section presents extensions to Markov predictors which include the aforementioned pattern matching. Markov predictors and its extensions are then later used as the basis of this chapter's prediction mechanisms.

7.2.1 Markov Chains

According to Grinstead and Snell [180] time and state discrete Markov chains are described by a set of states $S = \{s_1, \ldots, s_n\}$, where one state functions as the starting point of the process. In each time step, the process moves from one state s_i to another state s_j with the probability $p_{i,j}$ or stays in the same state s_i with probability $p_{i.i}$. Thereby, the probabilities $p_{i,j}$ or $p_{i,i}$ solely depend on the current state s_i and no previous states. These probabilities are called transition probabilities and can potentially be zero. All transition probabilities are combined into the transition matrix P. If P remains the same after each time step, the Markov chain is called time-homogeneous [181]. An extension to the classic Markov chain is provided by the concept of memory or a Markov chain of order m. There, a future state does not solely depend on the current state but rather on the

past m states visited. However, it is possible to translate a Markov chain of order m into a classic model, where a state s_i is defined as a ordered tuple of m states.

7.2.2 Markov Predictors

In general, predictors based on the Markov model are based on the idea that the next state depends on the last states [169]. Markov predictors have been deployed successfully in several fields of application as an online prediction mechanism without extensive training data like program phase behavior [169, 168, 170], memory prefetching [182], and branch prediction [183].

In [169], Sherwood et al. extended the basic concept of Markov predictors by run length encoding (RLE) because changes in program phase behavior usually occur after a sequence of stable phase behavior and RLE allows to reduce this stable phase sequence to a tuple of phase ID and occurrences in a row, i.e., the phase length. E.g., a stable phase sequence with phase ID 3 of $3, 3, 3, 3$ is compressed to the tuple $(3, 4)$. With the RLE concept, the phase history is compressed and the data that needs to be stored to make predictions reduced. To implement a RLE Markov predictor a hash table is used. The tag of the table is a hash value of tuples of phase ids and lengths. Next to this tag, the table stores the phase ID as prediction value. Figure 7.1 shows the implementation concept.

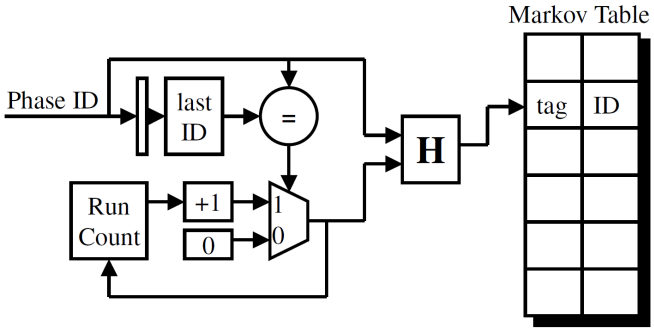


Figure 7.1: The implementation concept of the run length encoding Markov predictor by Sherwood et al. [169]

Chen et al. [183] combined Markov predictors with prediction by partial matching (PPM). The PPM algorithm of order m is comprised of $(m+1)$ Markov predictors, where a predictor of order j considers the last j states to make a prediction. If the Markov predictor of order m has no tag match for the sequence of the last m states, the $(m-1)$-th order predictor is checked and so on. Thereby, the 0-th order predictor just predicts the last seen state.

7.3 Prediction Mechanisms

In this chapter's introduction (s. Section 7.1) the two prediction scenarios that are considered in this chapter, independent tasks and dependent applications, are presented and motivated. As these scenarios pose different challenges, two separate prediction mechanisms are developed. Section 7.3.1 describes the mechanism used to predict new instances of independent tasks. The mechanism uses a prediction table inspired by the RLE Markov predictors by Sherwood et al. [169] (s. Sec. 7.2.2). In Section 7.3.2 the mechanism utilized to predict upcoming applications based on the recognition of dependency structures is elucidated. Again, a prediction table based on the RLE Markov predictor concept is employed. Similar to the work of Chen et al. [183], the prediction table is combined with PPM to filter out interfering parallel processes.

7.3.1 Predicting Independent Tasks

To predict upcoming instances of a task, a mechanism needs to be able to monitor the execution pattern of this specific task. In the context of runtime systems, tasks posses a unique identifier, e.g., an index, a user-given name, or a combination of both, which allows the mechanism to differentiate between tasks and to associate a pattern with a specific task. The profiling mechanism introduced in Sections 3.3.1 and 6.2 already monitors the execution time of a task instance by computing the difference between the starting time point st_i and finishing time point ft_i of the instance i. By additionally storing both time points for a task instance, it is possible to compute the time period $tp_{i,j}$ between to instances t_i, t_j of a task t. This information forms the basis to detect periodic behavior in the execution pattern of a task t.

The prediction mechanism uses a table similar to the Markov predictors presented in Section 7.2.2. The general structure of the table can be seen in Fig. 7.2. The table utilizes a unique hash ID as a tag to identify a task t. Next to the tag, the table stores the minimum tp_{min}, maximum tp_{max}, and average tp_{avg} time periods $tp_{i,j}$ between two instances t_i, t_j of this task t. Furthermore, the table also stores the number of instances that have been executed so far. These values are initially learned by monitoring during a training phase and then constantly updated during the remainder of the execution. A training phase for a specific task t is used to create confidence and stabilize the monitored values for this task, which increases the reliability of the mechanism's decisions and predictions. Only after the training phase for a task t is completed, predictions about the arrival of the next task instance t_i are made. This design decision is made to reduce the probability of false predictions as they can easily lead to bad system adaptations. In this work, the training phase for a specific task t is finished, when ten instances t_i of this task have been executed. Here, ten seems to be a good choice for the considered evaluation scenario as each task is repeated 500 times and ten repetitions therefore do not prevent the algorithm from making enough valid predictions but still make an observed

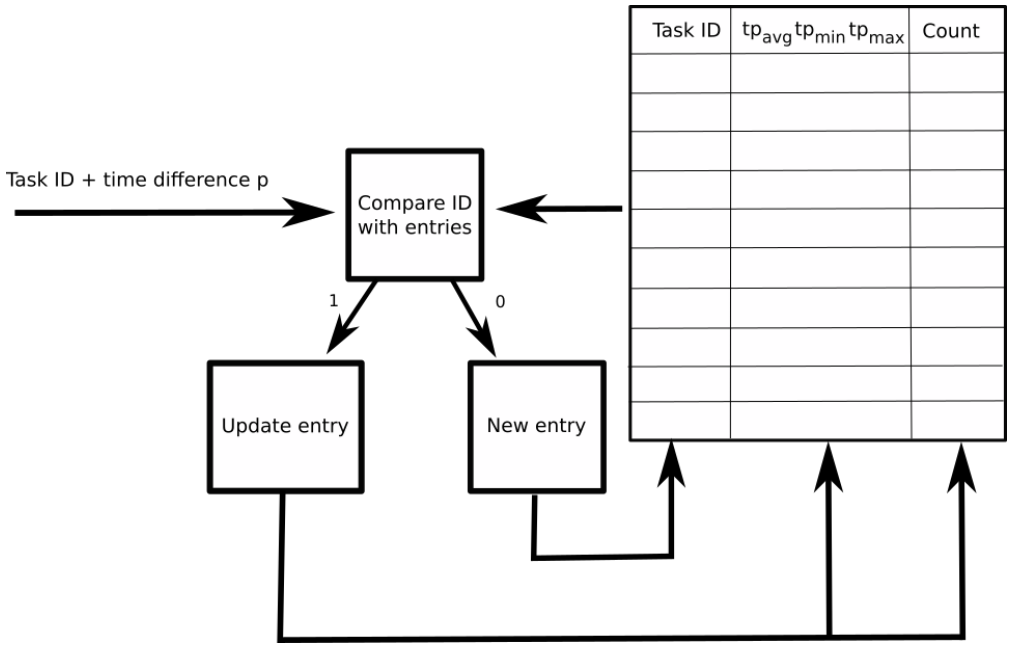


Figure 7.2: The general table concept used to predict upcoming instances of independent tasks

pattern statistically relevant. However, this limit is dependent on the application scenario and may vary in different scenarios.

Minimum and maximum time periods are used to differentiate between three execution patterns: periodic, sporadic, and aperiodic tasks. In this thesis, the assumption is made that sporadic tasks posses a minimum interarrival time tp_{min} and a maximum interarrival time $tp_{max} = l \cdot tp_{min}$ between two instances t_i, t_j of a task t, where l is a user-defined factor. In the formal definition of sporadic tasks [184], there is no upper limit for the interarrival time between two tasks. However, tasks that arrive arbitrarily cannot be predicted. Therefore, a compromise between the formal definition and predictability is made by limiting the maximum interarrival time. For this thesis, $l = 2$ is chosen. This is a reasonable choice for the intended evaluation scenario, as it allows for variation in the interarrival time of sporadic tasks to increase the difficulty of predicting their starting time while simultaneously allowing to distinguish sporadic from aperiodic tasks that may arrive randomly.

The variation between the minimum and maximum interarrival time is assumed to stem from an exponential distribution as the time period between two events in a homogeneous poisson process is exponentially distributed and poisson processes are often used to model the occurence of random events or tasks in a system, e.g., in queueing theory [185].

If for the stored values of a task t $tp_{max} > l \cdot tp_{min}$, where $l = 2$ in my evaluation scenario, is true, no further predictions are made and t is assumed to be aperiodic. A sporadic model is then assumed, if the difference $tp_{max} - tp_{min}$ is greater than $k \cdot tp_{max}$, where k is a user-defined factor $\in (0, 1 - \frac{1}{l})$ that is set to 0.1 in this thesis as the interarrival time of periodic tasks does not vary greatly and therefore a small threshold should be chosen. If both conditions are false, a periodic model is assumed. The assumed task model then implies how a prediction is computed. For an assumed periodic task, the stored average time period tp_{avg} is used to predict the arrival of the next task instance.

A sporadic task model, however, requires more complex computations. The stored minimum time period tp_{min} is utilized as minimum interarrival time. To predict the additional jitter, a value for λ, the parameter of the assumed underlying exponential distribution, has to be estimated. In statistics, the maximum likelihood method is deployed to estimate the parameters of a probability distribution. The maximum likelihood estimation of the exponential distribution returns $\lambda_{est} = \frac{n}{\sum_{i=1}^{n} x_i}$. This is the inverse of the observation mean $\bar{x}$, where $x_1, \ldots, x_n$ are samples of the distribution. The mechanism, however, does not directly monitor samples of the exponential distribution, i.e., the random jitter, but solely monitors the time period tp between two task instances. Therefore, the mean has to be computed another way. Computing the difference $tp_{avg} - tp_{min}$ results in the as of yet occurred average jitter jit_{avg}. The mechanism makes the assumption that the jitter is computed by multiplying an exponentially distributed variate with the minimum interarrival time tp_{min}, i.e., $jit_{avg} = \bar{x} \cdot tp_{min}$. Therefore, $\frac{1}{\bar{x}} = \lambda_{est}$ can be computed as follows:

$$\frac{1}{\bar{x}} = \frac{tp_{min}}{tp_{avg} - tp_{min}} = \lambda_{est} \tag{7.1}$$

With the estimate λ_{est} inverse transform sampling is used to create an exponentially distributed variate y_i. A prediction is then computed as follows:

$$\min(2 \cdot tp_{min}, tp_{min} + y_i \cdot tp_{min}) \tag{7.2}$$

7.3.2 Predicting Dependent Applications

Predicting applications or task sets based on dependencies requires the detection of these dependency patterns in past execution behavior. However, the focus of this thesis lies on parallel systems, where potentially multiple processes may run several applications and tasks in parallel with overlapping and independent dependency structures. Additionally, the runtime systems targeted here are located in user space and therefore are bound to a single process, i.e., a runtime system instance does not possess information about competing processes. Hence, there is no single instance with knowledge about dependencies between different processes.

In order to detect these dependency structures, a mechanism is required that observes the execution behavior and is able to detect repeating patterns while filtering out unrelated application executions. Next to predicting upcoming applications, the mechanism also has to predict the starting point of these applications. Thereby, I make the assumption that the time difference between dependent tasks remains constant.

To achieve these goals, a prediction mechanism based on Markov predictors and PPM (s. Section 7.2.2) is developed. The concept of this mechanism is show in Fig. 7.3. The central component of the mechanism is a prediction table comprised of a tag to

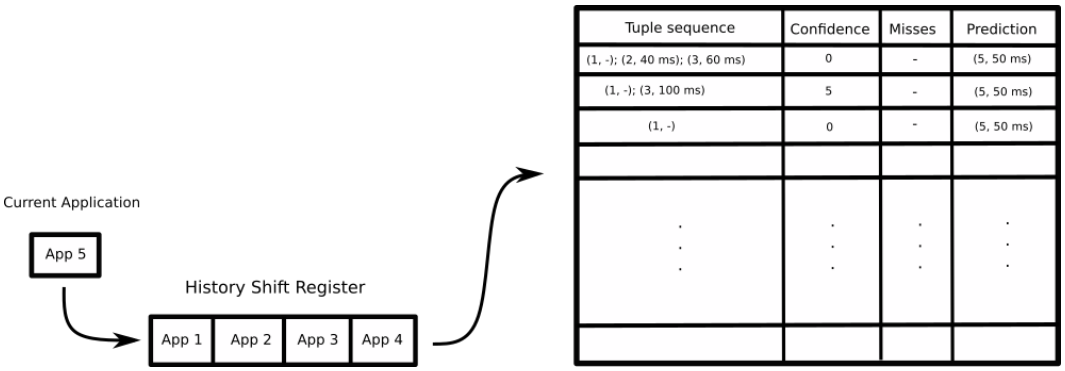


Figure 7.3: The mechanism concept to predict tasks or applications based on dependency structures

distinguish the table entries, a prediction value, and two status values. The tag consists of a set of tuples. Each tuple contains an application id i and the time period $\Delta\ tp_{i,j}$ between i and its successor application j in the tuple set, or just the application id in case of the first tuple. The tuple set hereby represents the application sequence that causes the execution of the predicted application that is stored as prediction value in the table. Besides the application id $a_{predict}$, the prediction value includes the time period $\Delta\ tp$ between $a_{predict}$ and the latest application in the tuple set, i.e., the first tuple. The two status values are counters. The confidence value counts the number of times the pattern has been observed up until now. Misses counts the number of times this entry mispredicted in a row. A correct prediction resets the miss counter.

In addition to the table, a shift register is used as an application history that stores the ids and the starting points of the m last executed applications. The history is employed to update the table and to determine the next prediction. Updating the table works as follows:

1. The latest application executed in the system functions as the prediction value of the potential table entries.

2. All potential table entry tags, i.e., the tuple sets, are computed by combining the

latest history entry with the power set of the remaining $m - 1$ history entries. The power set of the remaining entries is used as several dependency structures may overlap or independent applications may interfere with a pattern. An example computation is shown in Fig. 7.4.

3. The potential entries are sorted by decreasing tuple set size, e.g., an entry with a tag consisting of three tuples is listed before an entry with a tag of size two.

4. In this order, the created entries are compared to entries already included in the table. If there is a match, the confidence counter of the table entry is increased, the stored time difference $\Delta\ tp$ in the prediction value is updated by computing the new average, and the remaining potential entries are discarded. If there is no match, a new table entry is created and the next entry in the list is compared with the table.

The prediction table is updated each time an application execution finishes.

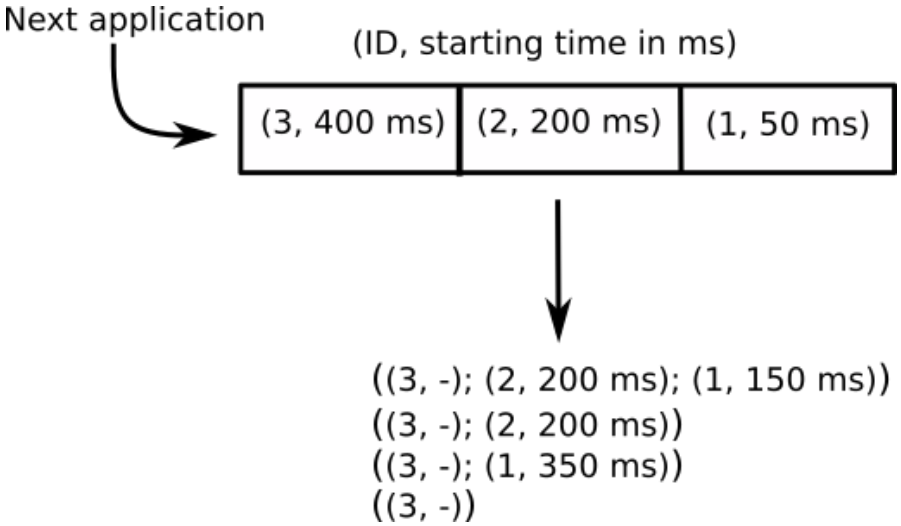


Figure 7.4: Computation of all possible entry tags with a given history of size three

After the execution of an application, a new prediction is computed as well. Again, all potential tags are computed as shown in Fig. 7.4 and then sorted by decreasing size. According to this sorting, the tags are compared with the table entries. If there is a match, the confidence value of this entry is checked. Only if the value is greater than a certain threshold, the prediction value of the table entry is used as prediction. To compute the starting point of the predicted application, the stored time period $\Delta\ tp$ is added to the finishing time of the latest application. If no match or an entry with a great enough confidence value exists, no prediction is made.

7.4 Evaluation

This Section presents the evaluation of the two prediction mechanisms developed in this chapter. Section 7.4.1 lists and discusses the results obtained for the mechanism predicting new instances of independent tasks. The results of the second mechanism, utilized to predict upcoming applications based on dependencies, are presented in Section 7.4.2.

To evaluate the mechanisms, simulations of task/application execution scenarios were used. Thereby, no real tasks/applications but rather a time series of task/application ids simulating an execution scenario were executed. As applications and task instances may run in parallel, a team of threads was deployed to start them at specific points in time. All simulations are executed ten times with differing random seeds.

The evaluation of the mechanisms requires a quantification of the prediction results. Therefore, an evaluation metric is needed. A metric that reflects the accuracy of regression experiments while trying to balance the influence of positive and negative mispredictions is $sMAPE$ (s. Eq. (6.3) in Sec. 6.4.2). However, in this section, I utilize a percentage based $sMAPE$, that compared to the definition used in 6.4.2 multiplies the computed value with $100\,\%$. The definition is as follows:

$$sMAPE = \frac{100\%}{n} \sum_{i=1}^{n} \frac{\|y_i - f_i\|}{(\|y_i\| + \|f_i\|)/2}, \tag{7.3}$$

where y_i is the correct value and f_i the prediction value. $sMAPE$ is applied to evaluate the prediction of future task instances as predicting the starting time of new task instances is the important aspect in this experiment.

In the second experiment, it is assumed that the time period between dependent tasks/applications remains stable. Therefore, the prediction evaluation can be reduced to a classification problem where accuracy (s. Eq. (6.2) in Sec. 6.4.2) can be applied as evaluation metric. However, compared to the evaluation problem in Section 6.4.2, here, no prediction is also a viable choice for the mechanism. Hence, accuracy is defined as follows:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}, \tag{7.4}$$

where TP and TN represent true positives and negatives, i.e., correct decisions of the prediction mechanism, and FP and FN false positives and negatives, i.e., wrong decisions.

7.4.1 Predicting Independent Tasks

The experiment conducted to evaluate the prediction mechanism for new instances of independent tasks was a simulation of the execution of ten tasks. Each task was associated with its own thread that started the instances of this task according to a specific

execution model. In total, the simulation was comprised of 500 instances of each task. The first ten instances of each task were used as a training phase for the prediction mechanism.

The task set was comprised of two aperiodic, two periodic, and six sporadic tasks according to the definition and assumptions made in Sec. 7.3.1. For the periodic tasks, the time difference between two task instances was computed by an exponential distribution. In general, the exponential distribution is defined by its probability density function $f(x)$ as follows:

$$f(x) = \lambda \cdot e^{(-\lambda x)} x \geq 0, \tag{7.5}$$

where λ is the parameter of the distribution, often called rate parameter.

Table 7.1 shows the parameter values chosen for the evaluation task set. As aperiodic tasks do not posses a minimum interarrival time tp_{min}, and periodic tasks have no additional random jitter, tp_{min} and λ, respectively, are not set for these tasks. Sporadic tasks, however, need values for both tp_{min} and λ.

Table 7.1: Parameter values of the task set used to evaluate the prediction mechanism for independent tasks

Task index	tp_{min}/tp	λ	model
1	–	5	aperiodic
2	50 ms	10	sporadic
3	100 ms	15	sporadic
4	150 ms	–	periodic
5	200 ms	25	sporadic
6	250 ms	30	sporadic
7	300 ms	35	sporadic
8	–	40	aperiodic
9	400 ms	–	periodic
10	450 ms	50	sporadic

To create variates for the exponential distributions needed for the aperiodic and sporadic tasks, inverse transform sampling was utilized. Inverse transform sampling creates variates of a distribution with a cumulative distribution function $F(x)$ by computing $F^{-1}(u)$, where u is uniformally distributed. The uniform distribution is generated by the 64 bit version [186] of Matsumoto's and Nishimura's pseudorandom number generator Mersenne Twister [187] with a random seed.

Next to $sMAPE$, the average estimated value λ_{est} to generate the random jitter for the sporadic tasks was also monitored. The results are shown in Table 7.2. For each sporadic task identified by the mechanism the obtained $sMAPE$ and average λ_{est} values, and for each periodic task identified only the $sMAPE$ value, are listed. As the

mechanism could detect both aperiodic tasks in all ten simulation runs and therefore did not make predictions for them, these tasks are not listed in the result table. The results

Table 7.2: Results of the prediction mechanism for independent tasks

Task index	$sMAPE$	λ_{est}
2	4.33 %	10.45
3	3.01 %	15.4
4	0.002 %	–
5	1.9 %	25.21
6	1.56 %	30.47
7	1.36 %	35.42
9	0.0004 %	–
10	0.9 %	50.39

show that in this scenario, the mechanism can correctly classify all ten tasks to their respective execution model, i.e., no predictions were made for the aperiodic tasks, and no jitter was added to the periodic tasks. The experiment also demonstrated that the mechanism can very reliably predict the period tp of periodic tasks. This is visible in the $sMAPE$ values $0.002\,\%$ and $0.0004\,\%$. As the highest $sMAPE$ value of the sporadic tasks is $4.33\,\%$ and the average of all six tasks is $2.18\,\%$, it is possible to claim that the time difference between two sporadic task instances can also be reliably predicted. This claim is additionally supported by the estimated λ_{est} values. At most, the average differs by $0.45\,\%$ from the value used to compute the exponential distribution.

7.4.2 Predicting Dependent Applications

The evaluation of the prediction mechanism based on dependencies was also conducted by simulating application executions. To simulate overlapping and interfering execution patterns, a set of threads was utilized. Each thread was associated with a specific execution pattern that is repeated 15 times. Between each pattern repetition, a pause of random length was inserted. The length of the pauses was again computed by an exponential distribution (s. Sec. 7.4.1). In total, the experiment was conducted ten times.

Four different patterns were used in this experiment. Between the dependent applications of an execution pattern, a constant pause tp_{const} was inserted. The first pattern is comprised of three applications, where the third application is dependent on the second, and the second application dependent on the first. The second pattern consists of two applications, where the second application is dependent of the first. The third and fourth pattern are comprised of a single application that is used to create interferences of the two regular patterns. The parameters employed to compute the pause lengths for all four

Table 7.3: Parameter values of the execution patterns used to compute pause lengths

Pattern index	tp_{const}	λ
1	300 ms	5
2	40 ms	0.75
3	– ms	4
4	– ms	0.25

patterns are listed in Table 7.3. For the first and third pattern, and the second and fourth, the exponential variates were multiplied by $1\,s$ and $0.5\,s$, respectively, to compute the length of the random pause between two pattern executions. To the pause of the third pattern an extra second was added.

In this experiment, a prediction was deemed to be correct if the predicted application id matches with an executed application and the predicted starting point does not deviate more than $0.5\,ms$ from the actual starting point of the application. The accuracy results of the experiment are shown in Fig. 7.5. As all patterns were executed 15 times and there are $2 + 1 = 3$ dependent applications, in theory there could be a total of $14 \cdot 3 = 42$ correct predictions. It has to be noted that in this experiment a confidence value of three was required, i.e., the mechanism had to observe a pattern three times, before a prediction was allowed. Hence, atleast $2 \cdot (2 + 1) = 6$ possible correct predictions cannot be made. Over all ten experiment runs, an accuracy of atleast 0.776 or $77.6\,\%$ was achieved. On average, the achieved accuracy amounts to $82.1\,\%$ with the highest accuracy being $86.7\,\%$. This shows that the mechanism resulted in a constantly fairly high accuracy despite the interferences and pattern overlaps.

To further dissect the results of the experiment, Table 7.4 additionally lists the total number of TP, FP, TN, and FN. The detailed results of Table 7.4 show that the mechanism does not predict applications that then do not arrive in the system. This is an important accomplishment, as wrong predictions may lead to erroneous system adaptions that result in disadvantageous system states. However, on average only $58.1\,\%$ of the possible predictions are made. In the worst case, only $47.6\,\%$ of the predictable applications are predicted correctly. As mentioned before, the mechanism requires repetitive observation of a pattern before a prediction is allowed. In this experiment, this reduces the number of possible correct predictions by six or $14.3\,\%$. To summarize, the mechanism minimizes the risk of mispredicting wile reducing the number of correct predictions. The fact that wrong predictions may lead to disadvantageous adaptations favors prediction mechanisms with such features.

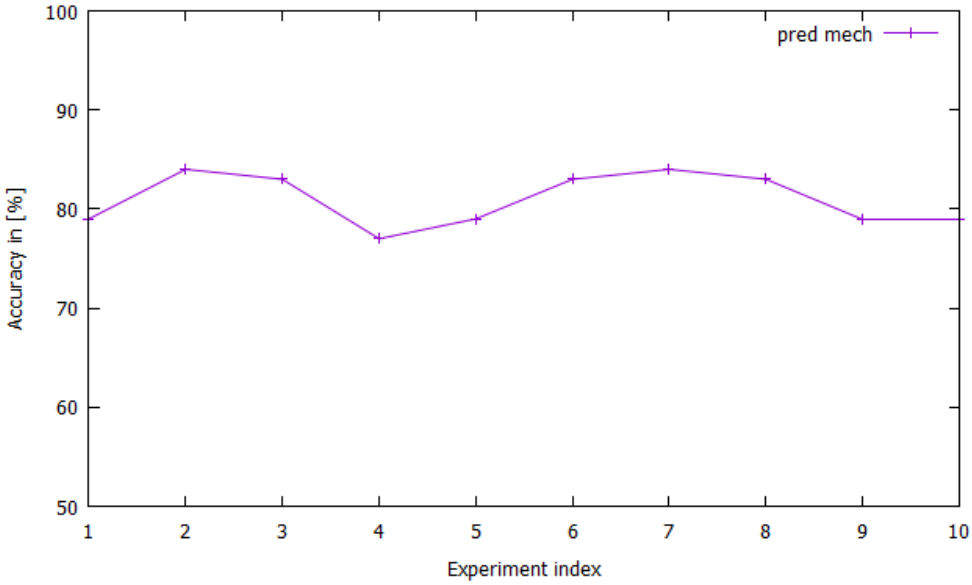


Figure 7.5: Accuracy of the prediction mechanism based on dependency patterns

7.5 Summary and Conclusion

Proactive behavior requires knowledge about the future. In the context of proactive system adaptation the required knowledge is comprised of future system states and environmental influences. Future system states in this thesis are influenced and determined by prospective tasks/applications and their associated execution costs, i.e., execution time, energy consumption, heat dissipation etc. Chapter 6 already discussed predicting the cost of tasks. Hence, the focus of this chapter lies on predicting upcoming tasks/applications. This chapter introduces two light-weight online prediction mechanisms that fulfill this task in the context of this thesis.

The first mechanism targets the prediction of future task instances and is based on RLE Markov predictors by Sherwood et al. [169]. For this work, the Markov predictor is extended with temporal information as there may be arbitrarily long pauses between task instances. The developed mechanism is made up of a table that stores the minimum tp_{min}, maximum tp_{max}, and average tp_{avg} of the time period $tp_{i,j}$ between two instances t_i, t_j of a task t as values, and a unique hash ID that uniquely identifies a task as the tag of the table entry. Based on these stored values, the mechanism differentiates between three possible task execution models: periodic, sporadic, and aperiodic tasks. In case a task is classified as aperiodic, no future instances of this task are predicted. For a periodic task, tp_{avg} is used to predict the starting time of the next instance.

If the mechanism classifies a task as sporadic, the computation of a prediction is

Table 7.4: Detailed prediction results for the dependency patterns experiment

Experiment index	TP	FP	TN	FN
1	22	0	56	20
2	29	0	56	13
3	26	0	56	16
4	20	0	56	22
5	22	0	56	20
6	26	0	56	16
7	27	0	56	15
8	26	0	56	16
9	22	0	56	20
10	24	0	56	18

more complex. Sporadic tasks have a minimum interarrival time tp that is extended by a random jitter. Thereby, the mechanism assumes that the random jitter is based on an exponential distribution. Via maximum likelihood estimation the parameter λ of the assumed underlying exponential distribution is estimated. By combining the estimated distribution with the monitored minimum interarrival time tp_{min}, the starting point of the next task instance is computed.

To evaluate the mechanism, the execution of a task set comprised of two aperiodic, two periodic, and six sporadic tasks was simulated. All tasks were executed a total of 500 times and the experiment was repeated ten times. The results show that the prediction mechanism is able to correctly classify all tasks. In addition, the mechanism is able to reliably predict the starting time of the upcoming task instances. For the periodic tasks, a $sMAPE$ value of $0.002\,\%$ respectively $0.0004\,\%$ was achieved. The average $sMAPE$ value obtained for the sporadic tasks was $2.18\,\%$.

The target of the second mechanism are dependent applications distributed over multiple processes. The idea of this mechanism is to utilize these dependency structures observed in past executions to predict upcoming applications. Again, a prediction table was used to store the necessary values for a prediction. Here, a prediction is comprised of an application id $i_{predict}$ and the time period $\Delta\ tp$ between $i_{predict}$ and and its predecessor application j. Besides the prediction value, two status values, a confidence and a miss counter, are stored in the table. The tag to identify a table entry consists of a set of tuples where each tuple contains an application id i and the time period $\Delta\ tp_{i,j}$ between i and its successor j in the tuple set, or just the application id i in case of the first tuple.

To observe execution patterns, the mechanism also applies a shift register that functions as an application history. As multiple processes may run in parallel, overlapping dependency structures and interfering applications have to be considered. To filter out

these interferences, the concept of PPM [183] is combined with the mechanism.

For the evaluation, again a simulation was utilized. A set of threads simulated overlapping and interfering execution patterns. Thereby, each thread was associated with a specific execution pattern. In total, four patterns were simulated and each pattern executed 15 times. The complete experiment was repeated ten times. Over all experiments, the mechanism achieved an average prediction accuracy of $82.1\,\%$. Particularly, the mechanism did not make false positive predictions, i.e., it did not predict applications that were not executed and it did not predict wrong starting times. This is an important feature for this thesis as wrong predictions may lead to disadvantageous adaptations and therefore disadvantageous system states. It is noteworthy, however, that minimizing the risk of wrong predictions also reduces the number of correct predictions.

To summarize, this chapter provides two online and light-weight prediction methods that enable the system to reliably predict upcoming tasks and applications. In combination with Chapter 6's mechanisms to capture the current system state and predict task costs, this creates the knowledge and builds the foundation necessary for proactive behavior and, in particular, proactive adaptations. The following chapter uses this knowledge to dynamically balance the contradicting optimization goals of the system.

Part III

Affecting Future System Behavior

CHAPTER

EIGHT

DYNAMICALLY BALANCING CONTRADICTING OPTIMIZATION GOALS

Chapter 6 and Chapter 7 provide a knowledge base for the runtime system of this thesis by monitoring and predicting the system state and environmental influences. This chapter utilizes this knowledge to proactively adapt the underlying system. In particular, the objective of this chapter is to find a suitable compromise between the contradicting optimization goals of the system. Thereby, each optimization goal is assigned a specific weight. The weights are used to form an evaluation function. With this evaluation function, scheduling decisions are made in the remainder of this thesis. In the thesis' holistic approach this chapter is represented by the blue components Controller and Offline Rule Generation.

The remainder of this chapter is organized as follows. The motivation for this chapter is provided in Section 8.1. Section 8.2 introduces the necessary theoretical background for the remainder of this chapter. This includes the formalization of multi-objective optimization, the mathematical concept of Markov decision processes, reinforcement learning, and the approach that is utilized in this chapter, learning classifier systems. Section 8.3 defines the problem that is the basis of this chapter. Related work is discussed in Section 8.4. Thereby, the focus lies on approaches that solve multi-objective scheduling problems. The solution approach of this chapter and its implementation are discussed in Section 8.5. For the solution mechanism, the multi-level observer/controller (MLOC) framework introduced by Müller-Schloer and Tomforde [9] to form organic computing systems is utilized. This section also introduces a reward function based on an execution cost simulator. Section 8.6 describes the experimental setup used to evaluate the MLOC framework and presents the results of the conducted evaluation. The chapter concludes with a summary and result discussion in Section 8.7.

8.1 Introduction

Today's computing systems are affected by an increasingly large set of optimization goals and constraints. Besides the optimization of applications' execution time, system objectives like the minimization of energy consumption, heat reduction, or system reliability and availability play gradually more important roles. The importance of these system objectives varies greatly depending on the system's field of application. Embedded systems with their limited resources and cooling capabilities have to put a bigger emphasis on energy and heat reduction in comparison to HPC or desktop systems. Additionally, specific embedded systems like cars or airplanes have strict safety constraints. Therefore, optimization objectives have to be adapted to the underlying system and its field of application.

Modern computing systems that operate in the fields of embedded systems or HPC, however, are highly dynamic in nature. Embedded systems allow users to install and delete applications in order to customize their devices according to their needs. Similarly, new and yet unknown jobs may constantly arrive in a high performance computing system.

Another factor that dynamically affects the importance of system objectives are environmental situations. A switched-off car is limited to its accumulator as a source of energy, hence creating a strict energy budget. An external heating source can lead to the overheating of the system, generating a system-wide need to reduce load and possibly switch off resources. Hence, computing systems cannot only be set up once at design time but rather have to be dynamically adapted and adjusted at runtime. This chapter provides a solution for this problem. It introduces a mechanism to dynamically balance contradicting system objectives by computing weights w_i that are utilized to form a system evaluation function $f(x) = \sum_i w_i \cdot x_i$. Hereby, x_i is the percental increase/decrease of system objective i compared to the best solution found so far.

Especially in embedded systems a fast response time is an important factor for a high-quality user experience, i.e., a proactive adaptation mechanism may not create extensive overhead that significantly slows down the response time. This means that exhaustive search algorithms may not be employed in these systems. Müller-Schloer and Tomforde [9] introduced a multi-level observer/controller (MLOC) framework that guarantees a fast response time while simultaneously exploring unknown situations and states offline, thus keeping the running system in a safe state. The framework is utilized in this chapter to create a proactive and dynamic adaptation mechanism designed for task scheduling. Particularly, this chapter makes the following contributions:

- A proactive and dynamic mechanism to balance contradicting system objectives based on Müller-Schloer's and Tomforde's MLOC framework is introduced. The mechanism utilizes a modified XCS into which a novel task execution simulator is integrated. This simulator is able to predict the total makespan, total energy

consumption, and the maximum processing unit temperatures. Task simulation is deployed to compute the necessary reward for the XCS.

- The mechanism is combined with a task scheduling algorithm to proactively determine new schedules according to upcoming situations.
- This approach reduces the makespan by $10.4\,\%$, the energy consumption by $4.7\,\%$, and the maximum temperature of the GPU by $3.6\,\%$ while only increasing the maximum CPU core temperature by $6\,\%$ in an evaluation scenario consisting of an application pattern that is repeated five times.

8.2 Theoretical Background

This section introduces the theoretical background necessary to understand the remainder of this chapter. First, finding a suitable compromise between contradicting system objectives, called multi-objective optimization, is discussed and formalized. Thereby, the definition of Pareto-optimality is presented. Section 8.2.2 elucidates the mathematical concept Markov decision processes, an extension of Markov chains introduced in Section 7.2.1. Markov decision processes are a formalization of sequential decision making and the problem of learning to achieve a goal and therefore are an important fundamental basis for machine learning concepts like reinforcement learning. Reinforcement learning is then discussed in Section 8.2.3. Finally, Section 8.2.4 explains Learning Classfier Systems and its extension XCS, the concepts the approach of this chapter is based upon. Learning Classifier Systems utilize a combination of genetic algorithm and reinforcement learning to solve learning problems.

8.2.1 Multi-objective Optimization

In multi-objective optimization problems several different objectives have to be optimized simultaneously [188]. This is in contrast to conventional optimization problems where only a single objective has to be optimized. The objectives of multi-objective optimization problems are usually contradicting, i.e., optimizing one objective worsens the other objectives, or else the problem could be reduced to a single-objective optimization problem.

Formally, the multi-objective optimization problem is defined as follows. The solution space of the problem is defined as a set of decisions D. A single solution x in the *decision space* Ω is a vector of n decision variables $x = (x_1, \ldots, x_n)$, called *decision vector* [189, 190, 191]. The quality of a solution is defined by M objective functions f_i that map a solution vector to a scalar value and together form the M-dimensional *objective space* Z. These functions have to be either minimized or maximized. Possibly, additional *constraints* $g_j(x) = b_j$ have to be fulfilled by solutions in order to be viable. All solutions satisfying the constrains constitute the *feasible solution space* S.

As the optimization criteria in multi-objective problems are typically conflicting, there usually does not exist a single optimal solution that minimizes/maximizes all M objective functions. Rather there is a set of solutions that represent optimal trade-offs between the different objectives, called *Pareto-set*. The objective function values of the Pareto-optimal set are called *Pareto-front*. Each solution in the Pareto-set is *Pareto-optimal*, i.e., there exists no feasible solution y that *Pareto-dominates* a solution x in the Pareto-front. Pareto-dominance is defined as follows [191, 189]:

A solution $x \in S$ *Pareto-dominates* (or short: *dominates*) a solution $y \in S (x \prec y)$ if
$$f(x) \neq f(y) \wedge \forall i : f_i(x_i) \leq_i f_i(y_i) \tag{8.1}$$

A solution $x \in S$ is *non-Pareto-dominated* (or short: non-dominated) by $y \in S$ if
$$y \not\prec x \tag{8.2}$$

8.2.2 Markov Decision Process

Markov Decision Processes (MDPs) are an extension of Markov chains introduced in Sec. 7.2.1. The following definitions and explanations are taken from "Reinforcement Learning: An Introduction" by Sutton and Barto [192]. The concept of MDPs is a formalization of sequential decision making and the problem of learning to achieve a goal. Thereby, an action does not only affect the immediate reward, but additionally influences subsequent situations or states, respectively and thus future rewards. A MDP consists of a learner, called *agent*, and its surrounding, called *environment*. Formally, a MDP is defined by the 4-tupel (S, A, p_a, R), where S is the set of potential states, A is the set of available actions, $p_a(s'|s, a)$ is the state-transition probability, where executing action a in state s at time t leads to state s' at time $t + 1$, and R the set of potential immediate rewards [193]. If S, A, and R have a finite number of elements such a process is called a finite MDP.

In a sequence of discrete time steps $1, 2, 3, \ldots$ the agent and the environment interact with each other. Fig. 8.1 shows the interaction cycle of a MDP. At each time step t, the agent receives a representation of the current state of the environment $s_t \in S$. The state s_t is used to select an action $a_t \in A(s)$, where $A(s)$ is the set of actions available in state s. In the next time step, the environment provides an immediate reward $r_{t+1} \in R \subset \mathbb{R}$ in part as consequence of the action. Additionally, the agent recognizes a new state s_{t+1}. The reward r_{t+1} and state s_{t+1} of the next time step $t + 1$ always only depend on the immediately preceding state s_t and action a_t and not on states or actions earlier in the past. Therefore, a requirement for states in MDPs is that they have to include all necessary information of the past interactions between agent and environment. A system where the states fulfill this requirement is said to have the Markov property.

In summary, the MDP framework abstracts the problem of goal-directed learning from interaction and reduces the problem to three signals between the agent and the environ-

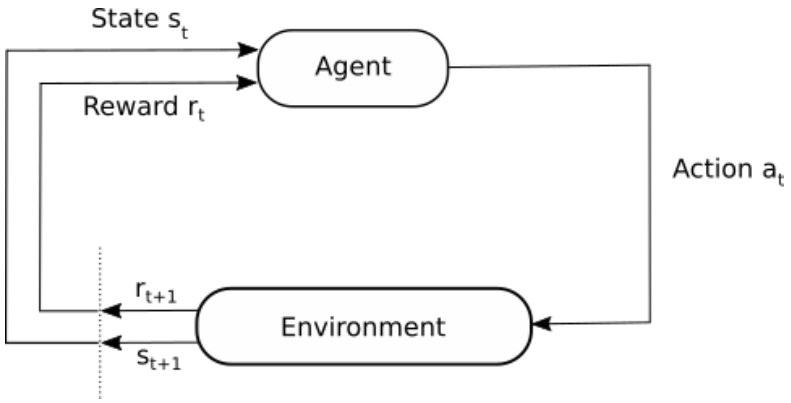


Figure 8.1: The interaction cycle between an agent and its environment in a Markov decision process [192, 131]

ment. The first signal represents the choices in form of actions made by the agent. The basis for theses choices, the states, are represented by the second signal. Rewards or the goal of the agent are defined by the third signal. The following section introduces reinforcement learning, a sub-category of machine learning. Reinforcement learning uses MDPs to formalize its problem statement and learning methodology.

8.2.3 Reinforcement Learning

Machine Learning is traditionally categorized into three main subclasses depending on the information or feedback available to the learning algorithm [192, 131]. The first two classes are supervised and unsupervised learning. In supervised learning, an exterior supervisor provides labeled training data that represents the correct and desired behavior, the label, in a specific situation [192]. The learning algorithm then has to generalize this knowledge to be able to act correctly in yet unknown situations. In contrast, unsupervised learning tries to find patterns and structures in unlabeled data without exterior knowledge [192].

Although labeled and unlabeled input seem to exhaustively classify machine learning paradigms, a third class, called reinforcement learning (RL), exists [192]. RL describes the phenomenon of learning by interaction and a feedback signal. It differs from supervised learning by the fact that the learning system learns from its own experience and not from an external knowledge source that provides examples of correct behavior [192, 131]. So, a RL system learns a task by reward and punishment without a distinct specification of how to correctly solve the task [193]. Generally, the goal of RL is to maximize a reward signal in contrast to finding hidden patterns or structures in a data set, the objective of unsupervised learning [131, 192].

RL problems may be episodic or continuous, i.e., the problem has terminal states, such as the outcome of a game, or is on-going. An example of an on-going task is an on-going process-control task. An idealized mathematical formalization of RL problems is given by MDPs (introduced in Sec. 8.2.2) [192, 194]. Consequently, a reinforcement learning system consists of the following elements [192].

- *Agent*: In RL systems the learner and decision maker is called agent. There is a strict distinction between the agent itself and its environment. Everything within an agent is controllable and known whereas the opposite may be true for its environment. The agent constantly interacts with its environment by selecting actions and tries to maximize a reward signal.
- *Environment*: Everything outside an agent is called environment. The environment may be not completely known to the agent. It reacts to the agent's actions by providing some information about itself, called state or situation, and a reward signal to the agent.
- *Policy*: The behavior of an agent is defined by its policy $\pi(a|s)$, i.e., the policy maps the perceived environment states to actions. In particular, $\pi(a|s)$ gives the probability that action a is selected in state s. A policy may be a simple function or lookup table, an extensive search, or even stochastic.
- *Reward*: The goal of the RL problem is maximizing the reward signal over the agent's life cycle. The reward is a numerical value $r_t \in \mathbb{R}$ provided by the environment after the agent executed an action. Thereby, the reward defines what is good or bad for an agent and causes policy changes. Generally, the reward signal is a stochastic function that maps an environment state and an action to a numerical value.
- *Value function*: The value of a state is the total reward an agent is expected to accumulate in the future starting from this state. Hence, compared to reward, which determines what is good in the short term, the value function defines what is good in the long run or the long-term desirability. Values are the basis of action selections as the objective is to maximize future rewards. In contrast to rewards, values are not directly provided by the environment, rather they have to be estimated from past observations made by the agent. Therefore, value functions are the main component a RL algorithm has to learn. Formally, a value function $v_\pi(s)$ under policy π is defined as follows:

$$v_\pi(s) = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s\Big] \text{ for all} s \in S, \tag{8.3}$$

  where γ is the discount rate, which is used to reduce the impact of future rewards r in a continuous problem, and t is any time step.

- *Model*: A model of the environment is utilized for planning, i.e., deciding on future actions, by making predictions about the behavior of the environment. Models are not mandatory for RL algorithms. Algorithms using a model are called model-based whereas algorithms without a model are called model-free.

In the next section, learning classifier systems are presented. Learning classifier systems are a class of learning algorithms that are tightly coupled to RL.

8.2.4 Learning Classifier System

The paradigm of learning classifier systems (LCSs) was introduced by John H. Holland in 1976 [195] as "a framework that uses genetic algorithms to study learning in condition/action, rule-based systems" [196]. Holland later revised the LCS framework to its current form [197].

In the context of LCSs, the system continuously interacts with its *environment*. The system determines the state of the environment s with a set of *detectors*, e.g., sensors that capture parts or the entire surrounding of the system [196]. The output of the detectors are converted into messages for the system and based on this messages the system selects its actions. These actions are then carried out by so called *effectors*, e.g., the muscles in a human body [196]. Choosing an action is carried out by a set of *rules*, called *classifiers*, that map messages representing states s_i to messages representing actions a_j. These rules are usually represented by an "IF condition THEN action" statement [198]. In the classic form of LCS, the condition is a string of the ternary alphabet $0, 1, \#$, where $\#$ stands for "don't-care" accepting every value. Hereby, an action is a string of the binary alphabet.

The general LCS framework utilizes two different components to drive its progress [199]. The first component is the discovery mechanism. In this context, discovery refers to discovering new rules in the solution space and adding them to the established rule set, called *population*. Discovery is done by a version of the genetic algorithm (GA). GA is a metaheuristic optimization algorithm belonging to the class of evolutionary algorithms. Introduced in 1975 by Holland [200], GA is based on Darwin's evolution theory of natural selection. Thereby, elements of the solution space are represented by chromosomes comprised of the *genome* (the rule condition) and the *phenotype* (the rule solution/action). GA iterates over a set of *chromosomes*, the *population*, until a termination criteria is satisfied. The population of a certain iteration is called *generation*. Fundamentally, GA is comprised of five steps:

1. An initial population of size N is created by generating usually random chromosomes.

2. All chromosomes of the current generation are evaluated by a fitness function.

3. Based on the computed fitness value, chromosomes are selected to create offspring. In the literature, there are several different selection methods, e.g., roulette wheel selection, and tournament selection, that usually include some randomization. Additionally, if the population has grown larger than N, chromosomes are deleted based on their fitness.

4. The selected chromosomes are used to create offspring by utilizing the genetic operators *crossover* and *mutation*.

   - Crossover: Two parent chromosomes are split up and then combined to produce offspring chromosomes.
   - Mutation: A parent chromosome is randomly mutated, i.e., parts of the values are randomly altered,

5. The steps two to four are repeated until a termination criteria is fulfilled.

In the literature, there exist several modifications and variations of the genetic algorithm and the aforementioned steps. For example, different methods of chromosome selection, and various implementations of the genetic operators are employed depending on the field of application and the underlying problem.

The second component is the learning mechanism of LCS. The objective of the learning mechanism is to improve local performance by "tuning the associated statistic/parameters of a rule through accumulated trial-and-error experience" [198]. Therefore, one or more strength or accuracy fitness values and parameters are associated with each classifier. These values are iteratively updated by either reinforcement learning (RL) (s. Sec. 8.2.3), i.e., reward/punishment of past interactions, or supervised learning, i.e., training examples of correct behavior [199]. Traditionally, LCSs use RL and a strength-based value that estimates the expected payoff of the associated classifier.

Figure 8.2 shows the fundamental flow diagram of the basic LCS algorithm with RL. A typical algorithm iteration consists of the following nine steps:

1. One or several detectors capture the current state of the environment.

2. A representation of this current state encoded with a particular alphabet is provided to the LCS, in particular to the classifier population $[P]$. $[P]$ has a limit N for its size set by the user.

3. The LCS compares all classifiers with the current state, a process that is called *matching*. All classifiers which conditions match the current state form the match set $[M]$.

4. If $[M]$ is empty or does not contain enough classifiers, either in total or for a specific action a, again a limit is set by the user, *covering* is activated. Covering, then, generates new classifiers that cover the input state.

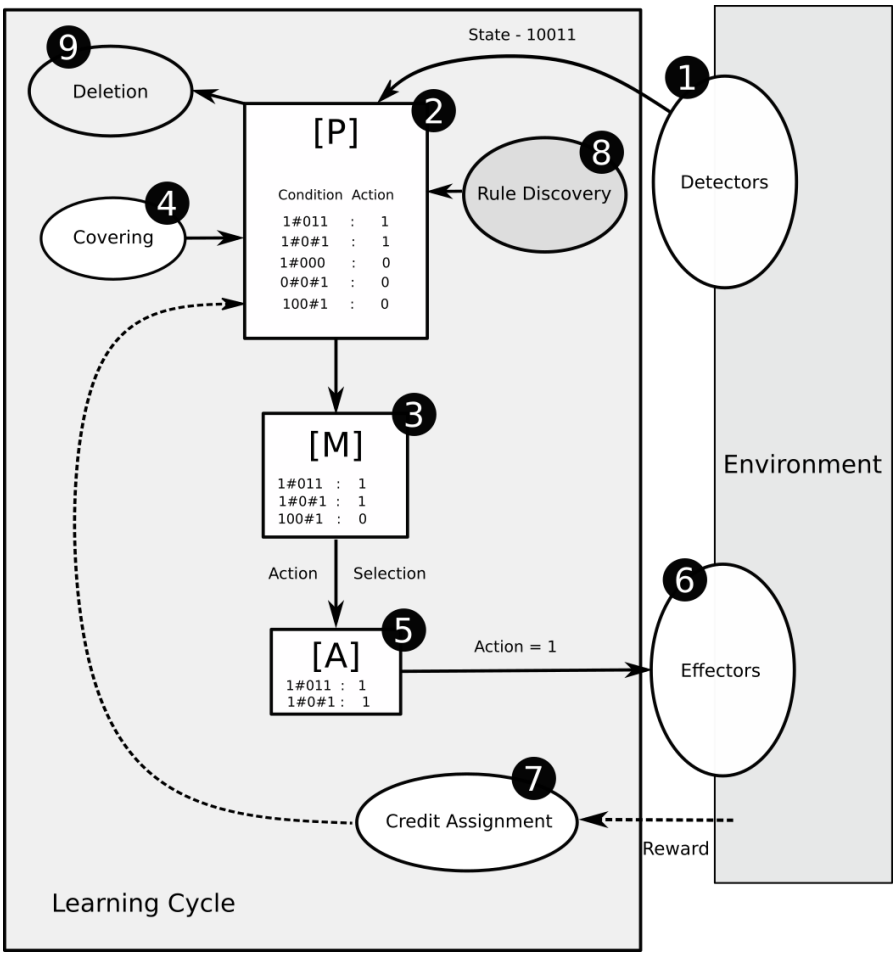


Figure 8.2: The fundamental flow diagram of a LCS with RL [199]

5. Usually $[M]$ is comprised of classifiers with different action phenotypes. Therefore, one of the available actions has to be selected. Thereby, different strategies can be utilized and a trade-off between exploration and exploitation has to be considered. Typically, the selection is based on the fitness value of the classifiers and involves some sort of randomization. All classifiers with the selected action a_{sel} then form the action set $[A]$.

6. In the next step, the selected action is executed by a set of effectors. AS a result of the action, the environment gives feedback in the form of a reward signal.

7. The LCS uses the reward signal to update the parameters of the classifiers whereby all classifiers in $[A]$ are updated. Again, different update strategies are available depending on the parameters of the used LCS implementation.

8. *Rule discovery* is used to create better classifiers by applying the genetic algorithm (GA). Here, LCS implementations differ in the time at which rule discovery is invoked, and in the implementation of GA.

9. To make room for the newly generated classifiers by GA, old classifiers have to be deleted. Like action selection, *deletion* typically is based on the classifier fitness and involves some sort of randomization. The specific deletion algorithm is thereby dependent on the LCS implementation.

As mentioned before, LCS includes several parameters and building blocks that can be tuned and varied by the user. The following section introduces a prominent variant of LCS, called XCS.

XCS

In 1995 Stewart W. Wilson introduced a LCS modification, called XCS [201]. Classic LCSs utilize a strength-based value as a prediction of future reward and as fitness for GA. This means that a classifier that predicts a small payoff is less likely to survive a GA iteration than a classifier that promises a bigger payoff. However, the amount of payoff a classifier receives may not adequately represent how accurate its prediction is. It is quite possible that a classifier with a small future reward prediction accurately represents the outcome of this situation. Additionally, a classifier with small reward may potentially still be the best solution for this particular environmental niche and should therefore remain in the population.

To solve this problem, Wilson introduced a separate fitness value for GA next to the strength of the classifier utilizing knowledge gained in the research of RL. The GA fitness is hereby based on the accuracy of the prediction and not the prediction value itself [201]. More precisely, the following three parameters replace strength in XCS:

1. *Prediction* p: the average received future reward

2. *Prediction error* ε: "an average of a measure of the error in the prediction parameter" [201]

3. *Fitness* F: the inverse of the prediction error

Next to the separation of strength, GA is only applied to the matching set $[M]$ rather than to the whole classifier population $[P]$, thereby reducing survival competition between different environmental niches.

In total, the following modifications were introduced with XCS over the whole algorithm iteration:

- Action selection: To select an action, a prediction array with system prediction $P(a_i)$ for every action a_i is computed. The computation of $P(a_i)$ utilizes a fitness-weighted average of the prediction values of the classifiers advocating action a_i.

- Update: The update function computes new values for the three parameters prediction p, prediction error ε, and fitness F of the action set classifiers in the last time step $[A]_{-1}$. The update computation itself is based on Q-learning [202] and is defined as follows:

  1. $F_j \leftarrow F_j + \beta(\kappa'_j - F_j)$, where β is the user-set learning rate,
     $\kappa' = \dfrac{\kappa}{\sum_{cl \in [A]} \kappa_{cl}}$, and
     $\kappa = \begin{cases} 1 & \text{if } \varepsilon_j < \varepsilon_0 \\ \exp\left(ln(\alpha)\frac{\varepsilon_j - \varepsilon_0}{\varepsilon_0}\right) & \text{else} \end{cases}$,
     where α and ε_0 are user-set parameters $\in (0, 1)$
  2. $\varepsilon_j \leftarrow \varepsilon_j + \beta(|r_j + \gamma \max_a P(a_j) - p_j| - \varepsilon_j)$, where γ is the user-set discount rate, and r_j is the reward received in the last time step
  3. $p_j \leftarrow p_j + \beta(r_j + \gamma \max_a P(a_j) - p_j)$

- Discovery component: The GA is applied to the match set $[M]$ and not the whole classifier population $[P]$. The GA is activated for a match set if a certain number of time steps has passed since the last GA iteration in that match set.

- Numerosity: If a newly added classifier has the same genome and phenotype as an existing one, the classifier is not added, but a *numerosity* n_{num} field is incremented by one in the existing classifier. A newly added classifier is initialized with a numerosity value of 1.

In additional publication, Wilson further extended and improved XCS. The input of an XCS was extended to continuous real-values represented as intervals with a center and deviation value [203]. This extension also required adaptations in the mutation operator and the covering mechanism. To improve the generation of new classifiers, Wilson restricted the application of GA to the action set $[A]$ instead of the match set $[M]$ in the original XCS version as the combination of classifiers with different actions often resulted in inaccuracies [204]. Additionally, a subsumption rule was introduced in the generation of offspring [204]. If an offspring is created whose condition is logically subsumed (all strings that match the offspring's condition are a subset of the match set by the parent) by one parent and the parent is accurate and sufficiently experienced, the offspring is discarded and the numerosity of this parent increased by one. In 2002, Wilson introduced XCSF [205], an extension to XCS that allows to learn functions.

8.3 Problem Statement

The focus of this chapter is solving the multi-optimization problem of finding a suitable balance between the differing system objectives. Thereby, the underlying system is assumed to be dynamic. In the context of this thesis, dynamic means that new applications or tasks and task instances may arrive at any time in the system, and unforeseen environmental events may occur.

This thesis focuses on four system objectives: makespan, i.e., the total time that is needed to execute all applications, energy consumption, i.e., the amount of energy that is required to execute all applications, system temperature, and system and component reliability. Reliability is considered implicitly in this thesis. An application is allowed to set a minimum reliability percentage for its execution. The symptom-based fault detection mechanism introduced in Chapter 6.3 enables the system to compute a heuristic reliability metric. Combined with the required minimum reliability percentage of the application, it is possible to compute how often an application has to be repeatedly executed on a specific processing unit to fulfill its minimum reliability requirement. The additional executions are reflected in increased execution costs for the application, e.g., increased total execution time.

The balance of the system objectives is expressed through normalized weights $w_i \in \mathbb{R}$, where $\sum_{i=0}^{n} w_i = 1$, that shall be dependent on the current state of the system and the environment, and the upcoming tasks to be executed. Therefore, the optimization problem has to map an input vector $\vec{s} = (s_0, \ldots, s_m)$ that represents the current situation through different metrics s_j to a vector $\vec{w}$ representing the weights of the system objectives.

These weights are utilized to build an evaluation function $f(x)$ for task scheduling algorithms, in particular list scheduling. Hereby, x is a vector $\vec{x} = (x_0, \ldots, x_n)$ where x_i is the percental improvement/deterioration of objective i, e.g., the makespan over all scheduled tasks, compared to the best solution found so far. The evaluation function f is then defined as $f(x) = w_1 \cdot x_1 + \cdots + w_n \cdot x_n$.

Determining continuous weights for the system objectives translates to a regression problem. As regression problems are computationally more complex than classification problems, I reduced the weight mapping to a classification problem by introducing a step size of $10\,\%$ instead of continuous values for the weights. Hence, there is only a numbered amount of possible weights that can therefore be interpreted as classification categories.

The systems considered in this thesis are continuously running and new applications or tasks are allowed to dynamically arrive in the system at any time. Thereby, past scheduling decisions may affect future decisions because they partly determine the system state the next scheduling decision is based upon. However, it can generally be assumed that knowledge about the current state is sufficient to make a decision. Therefore, the states fulfill the Markov property and the scheduling problem can be regarded as

a Markov decision process. In the scope of RL, a problem without a definite end is called a *continuous task* and a task where a goal has to be achieved over several state-action transitions is called a *multi-step task*.

If the problem of balancing the system objectives is viewed as a RL problem, the set of input vectors s_k that represents the current situation and the set of weights $\vec{w}$ are synonymous with the state set S and the action set A in the formal RL definition, respectively. A model of the environment can be provided by the monitoring component and history database as it allows predictions and estimations about the effects of different schedules on the system. So, in summary, the problem of this chapter can be expressed as a continuous, multi-step, multi-objective RL problem whose states fulfill the Markov property and therefore can be interpreted as a Markov decision process and solved accordingly.

8.4 Related Work

In the literature, there exist several approaches and algorithms that target considering multiple optimization goals in the context of task scheduling. Particularly, energy-aware scheduling algorithms, i.e., algorithms that try to minimize the energy consumption next to minimizing the makespan, are widely studied. Lee and Zomaya proposed the two energy-conscious scheduling heuristics ECS and ECSidle [206] that utilize dynamic voltage scaling (DVS) to adjust processors' voltage supply levels (VSLs) in order to reduce energy consumption. The scheduling decisions of these two heuristics are based upon a new metric called relative superiority (RS). For a given task t_i, a processor p_j, and a VSL $v_{j,k}$ with the as of yet best task processor combination t', p', RS is defined as follows:

$$\begin{aligned} RS(t_i, p_j, v_{j,k}, t', v') = -\Bigg(& \frac{E_d(t_i, p_j, v_{j,k}) - E_d(t_i, p', v')}{E_d(t_i, p_j, v_{j,k}} \\ & + \left(\frac{eft(t_i, p_j, v_{j,k}) - eft(t_i, p', v')}{eft(t_i, p_j, v_{j,k})} \right) \\ & - \min est(t_i, p_j, v_{j,k}), est(t_i, p', v') \Bigg), \end{aligned} \tag{8.4}$$

where eft and est are the earliest finish and earliest start time, respectively. The algorithm chooses the task processor pair that maximizes the RS value. Additionally, the makespan-conservative energy reduction (MCER) technique is applied after a schedule is computed to further check for possibilities of energy reduction without increasing the overall makespan.

As longer processor idle times are not considered in this approach, the RS metric was adapted to:

$$RS_{idle} = \begin{cases} -\left(\frac{E_d(t_i,p_j,v_{j,k}) - E_d(t_i,p',v')}{E_d(t_i,p_j,v_{j,k}} + \frac{eft(t_i,p_j,v_{j,k}) - eft(t_i,p',v')}{eft(t_i,p_j,v_{j,k}) - est(t_i,p_j,v_{j,k})}\right) & \text{if } eft(t_i,p_j,v_{j,k}) < eft(t_i,p',v') \\ -\left(\frac{E_d(t_i,p_j,v_{j,k}) - E_d(t_i,p',v')}{E_d(t_i,p_j,v_{j,k}} + \frac{eft(t_i,p_j,v_{j,k}) - eft(t_i,p',v')}{eft(t_i,p',v') - est(t_i,p',v')}\right) & \text{otherwise,} \end{cases} \tag{8.5}$$

which favors shorter task completions. The MCER technique is also modified to include idle energy consumption. Compared to this thesis, the weighting between makespan reduction and energy consumption minimization is static and not dependent on the current situation.

Chen et al. utilize dynamic voltage and frequency scaling (DVFS) to reduce energy consumption of iterative workloads with parallel tasks [207]. Their proposed algorithm adjusts the CPU core frequencies of cores that execute smaller workloads, thereby reducing energy consumption without degrading performance. An approach that targets CPU-GPU architectures is provided by Ma et al. [208]. Their approach, called GreenGPU, is separated in two tiers. The first tier consists of a workload balancing mechanism that distributes iterative workloads to the CPU and GPU. Hereby, the mechanism tries to reduce idle times of both processors. The second tier adjusts the frequencies of the CPU and GPU according to their utilization. If low utilization is monitored, the frequency is reduced. Both of these approaches only target specific workloads and only consider energy optimization when it does not affect performance. Additionally, preparing a workload in such a way that it is finely distributable over a GPU and CPU is a difficult task.

In [209] four heuristics based on the Min-Min algorithm (s. Sec. 9.2.3) that consider both energy and makespan are presented. The first heuristic uses Min-Min to compute a task schedule and then utilizes dynamic voltage scaling (DVS) to reduce the energy consumption without violating a predefined schedule length L_c. As finding the optimal voltages is a computationally complex problem, the second heuristic simplifies the problem by restricting the voltage selection to two voltage levels v_l and v_h that satisfy $v_l \leq \frac{ft(p_j)}{L_c} v_{max} \leq v_h$, where $ft(p_j)$ is the finish time of processor p_j. The third heuristic schedules the tasks in order of the Min-Min ranking. After each scheduling decision, the completion time of the task is checked and if it is below a certain threshold, the supplied voltage will be reduced to reduce energy consumption while not increasing the completion time too much. The last heuristic, again, schedules each task in the same order,

however first minimizing its energy consumption while maintaining a makespan threshold. Like the other approaches presented before, these heuristics always find a static compromise between the two objectives makespan and energy and do not dynamically adapt to new situations.

Another optimization objective that is combined with makespan minimization is the system temperature. In [210], a task scheduling algorithm for soft real-time tasks on homogeneous chip multiprocessors that tries to minimize the core temperatures while adhering to task deadlines is introduced. The work utilizes the compact RC thermal model and pre-built look-up tables to compute the thermal distribution on the chip after a task allocation based on the power input on each core. The scheduling algorithm considers all idle cores as potential candidates for a newly arrived task and computes the thermal map for each potential mapping candidate. A task is finally assigned to the core that minimizes a weight $w = \sum_{i,j} tm(i,j) \cdot rrt(i,j)$, where i, j are the processor core indices, tm is the thermal map, and $rrt(i,j)$ is the remaining runtime of the task on core (i,j).

In their work, Sheikh et al. additionally consider both the system temperature and energy consumption as optimization objectives next to makespan [211]. Rather than optimizing only one objective under constraints of the other two objectives, Sheikh et al. offer an algorithm that computes the Pareto front of the solution space. To compute the Pareto front, their algorithm, called E-FORCE, utilizes SPEA-II, an evolutionary technique that deploys genetic operators, to create new solutions. The user can provide a preference vector to select a solution out of the Pareto front. My approach selects weights for the different system objectives according to the current situation and does not require the computational overhead of an evolutionary algorithm each time a scheduling decision is needed.

To proactively adapt the contradicting optimization goals, this chapter deploys Organic Computing (OC) principles, in particular a hierarchical observer/controller (O/C) architecture that includes an XCS (s. Sec. 8.2.4) component. Such an approach has been used in different fields of application in the literature. Prothmann et al. [212] introduced an Organic Traffic Control (OTC) system that utilizes OC mechanisms to self-organize and self-adapt urban traffic. This system consists of three parts, a self-adaptive traffic light control mechanism, a self-organized coordination mechanism to obtain a Progressive Signal System (PSS), and self-organized route guidance. The traffic light control mechanism employs a hierarchical O/C architecture to determine the length of the green light phases of an intersection. Thereby, the first layer including a modified XCS (s. Sec. 8.5) matches current traffic conditions to its rule base selecting a traffic light configuration. Unknown traffic situations are handled offline by the second layer that utilizes an evolutionary strategy to create new rules and then determines their value in a simulation. The feedback for a selected traffic light configuration and thus the rule reward is the

average delay of the complete intersection. Proactivity for the OTC system is achieved by the self-organized route guiding mechanism, called Dynamic Route Guiding (DRG) mechanism, that recommends the best path trough the road network to drivers at each intersection.

The hierarchical O/C concept is also deployed by Hurling et al. [213] to form an Organic Network Control (ONC) system. Again, the first layer uses a modified XCS to match situations with a rule base where a rule selects the parameters of the underlying communication protocol. As trial-and-error runs are not feasible in a technical system, a second offline layer is also utilized to generate new rules and simulate their value with a network simulator. This concept has been implemented for Mobile Ad-Hoc Networks (MANets) [214], where the ratio of forwarded to received messages is optimized, Wireless Sensor Networks (WSN) [215] optimizing the number of active nodes, the coverage of the environment, and energy consumption, and finally Peer-to-Peer (P2P) protocols [214] maximizing the received download rate.

8.5 Approach and Implementation

This thesis' target systems and fields of application entail specific requirements for a dynamic system adaptation mechanism. The mechanism has to make fast decisions without large computational overheads compared to the actual workload in order for the system to be able to benefit from the adaptations. Furthermore, trial-and-error exploration runs, i.e., evaluating unknown adaptations at runtime, need to be avoided as they may lead to dangerous behavior in safety-critical systems or may drastically reduce quality of service and therefore result in a bad user experience. Still, system adaptations are necessary as the target systems are dynamic in nature and allow updates and the addition of new applications.

The multi-level observer/controller (MLOC) framework introduced by Müller-Schloer and Tomforde [9] provides a solution to these challenges. Figure 8.3 gives an overview of the abstract framework. The basic framework consists of four layers, the productive layer, the reactive adaptation layer, the reflection layer, and the collective layer. On the lowest level, the productive system encapsulates the system that needs to be controlled and adapted. This system is continuously observed by the higher layers. Layer 1, the reactive adaptation layer, is the first control mechanism of the framework. Therefore, it observes the underlying system and deploys a modified XCS (s. Sec. 8.2.4) to proactively adapt the underlying system to the current situation. The most important modification is outsourcing the generation of new rules typically done by a genetic algorithm to the third layer. Separating the generation of new rules from the XCS allows the framework to compute new rules offline and in parallel to the running system. In order to still keep the system running, a fall back covering mechanism and backup default rules are utilized in the absence of rules for a specific situation.

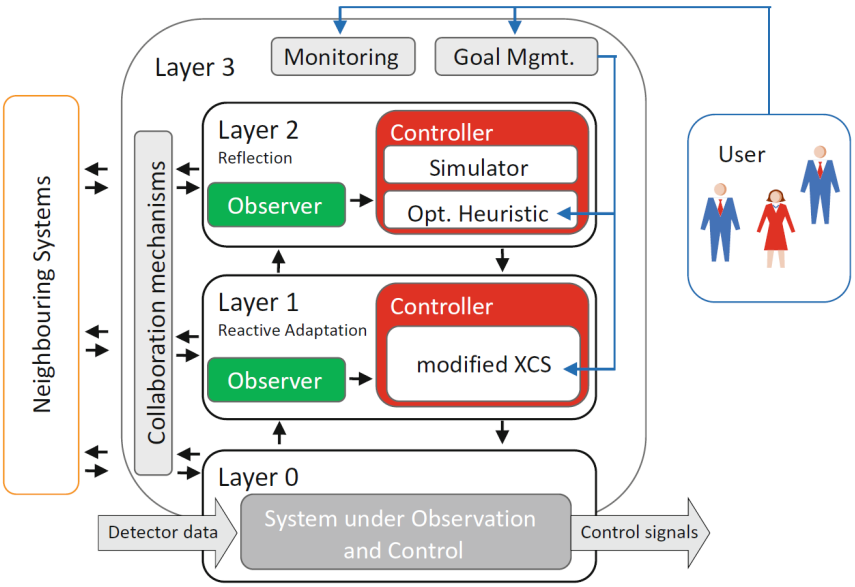


Figure 8.3: The abstract multi-level observer/controller (MLOC) framework by Müller-Schloer and Tomforde [9]

The reflection layer observers layer 1 and in the case of a missing rule, the process of generating a new rule is started. As the computation has to be done offline and may not affect the actual system, a simulator is deployed. To compute new rules, an evolutionary algorithm is used. The last layer provides an interface to communicate with neighboring systems and the user, e.g., to allow the user to influence the system objectives.

In the following sections, the implementation of this model in this thesis is elucidated in detail. Section 8.5.1 discusses the implementation of the XCS that utilizes the knowledge provided by Chapters 6 and 7, i.e., the observer component of the reactive layer, to proactively adapt the underlying system.

8.5.1 Implementation of the Modified XCS

The XCS implementation of this thesis is based on the C++ library XCSLib [216]. The basic functionality of XCSLib was ported from C++ to C for the integration into the runtime system HALadapt. Compared to the definition of an XCS in Sec. 8.2.4 two changes are made. Next to the three strength values prediction p, prediction error ε, and fitness F, each rule stores the average size of its match sets $size_{ms}$. $size_{ms}$ is updated as follows:

- $size_{ms} += \gamma \cdot (size_{ms_{cur}} - size_{ms})$, where $size_{ms_{cur}}$ is the size of the current match set.

Additionally, the formula to compute κ is changed to:

- $\kappa = \begin{cases} 1 & \text{if } \varepsilon_j < \varepsilon_0 \\ \alpha(\frac{\varepsilon_j}{\varepsilon_0})^{-\nu} & \text{else} \end{cases}$,
  where ν is usually set to 5.

For the communication with HALadapt, the XCS interface includes two functions, *receiveInput()* and *receiveReward()*. *receiveInput()* takes the description of the current system state and computes the according system objective weights. The system state description is comprised of the following metrics:

- A description of the current task graph
  - The total number of tasks
  - The average execution time of the longest graph path
  - The maximum graph width, i.e., the maximum number of tasks that may be executed in parallel
- A description of the hardware state
  - The current temperatures of the available processing units
  - The average waiting time over all processing unit queues
- Given constraints, e.g., maximum temperatures for the system's processing units
- A description of the next prospective task graph if a prediction is possible

Fig. 8.4 depicts the computational sequence of *receiveInput()*. First, the match set, i.e., all rules that match with the given system state description, is determined. If not enough rules for the given state exist, the offline rule generation mechanism is started. This mechanism simulates several task schedules created with different system objective weights and selects the weight with the best result as the new action. In this thesis, the system objectives total makespan, total energy consumption, and maximum processing unit temperatures are considered. The reliability of processing units acts as a constraint that determines how often a task has to be repeated on a certain processing unit to achieve the desired result reliability. The number of task repetitions hereby depends on the current value of the processing unit's heuristic fault rate (s. Ch. 6.3 for more details). The offline rule generation employs a step size of 0.1 for the weights of the system objectives.

If there is no single task schedule that dominates all other schedules, i.e., a Pareto set of best solutions exists, a solution of the Pareto set is randomly selected. To finally create the new rule, the received system state description is randomly widened. A lower

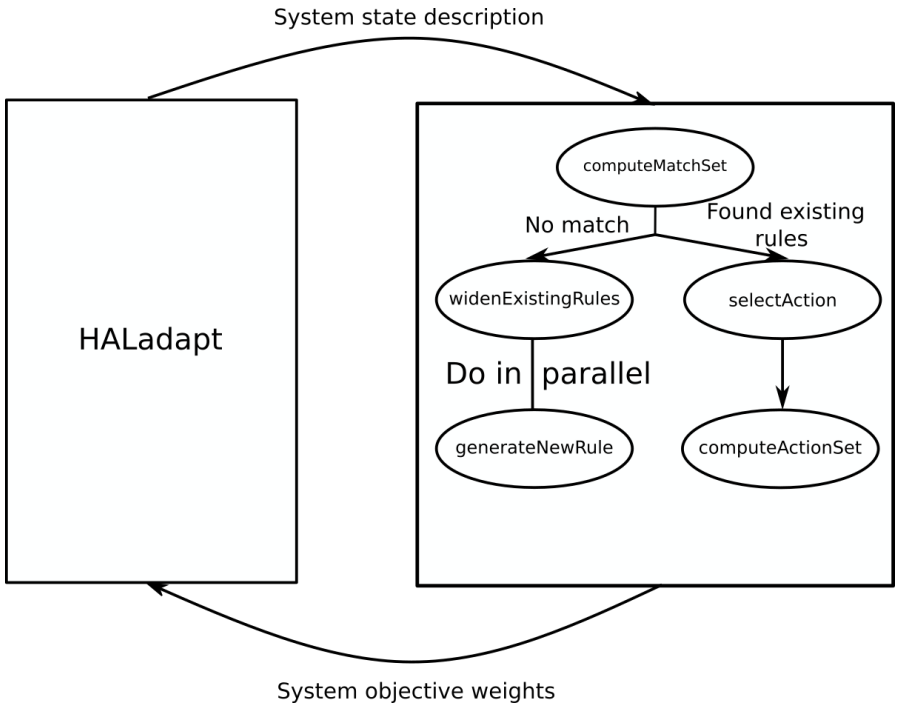


Figure 8.4: Computational sequence of *receiveInput()*

and upper bound are created by decreasing/increasing each metric value by 0 to $50\,\%$ at random. At last, the new rule is added to the XCS. If, after adding a new rule, the total number of rules exceeds a user-set maximum value, a roulette-wheel deletion mechanism is started. Here, every rule computes a deletion probability $p_{delete} = size_{ms} \cdot n_{num}$. If a rule has an experience value larger than ϑ_{delete} and a fitness value F smaller than $\delta_{delete} \cdot \overline{F}$, its deletion probability is increased to $p_{delete} = \frac{\overline{F}}{F}$. A random number draw finally decides which rule is deleted.

As the rule generation is executed offline, i.e., in parallel to the running system, the system cannot be stopped while the new rule is computed. In case there is at least one rule that satisfies the system state, this rule is utilized to keep the system running. If no rule exists yet, it is checked whether it is possible to widen an existing rule or, if this is not possible, balanced weights are returned as a fallback rule. A rule may be widened if its Euclidean distance is smaller than a specified threshold.

Otherwise, if there exists at least δ_{match} rules that satisfy the system state, the action selection algorithm is started. This algorithm either selects the action with the best predicted payoff or a random action of the match set. The decision between best and random selection is made by generating a random value between 0 and 1. If the value is smaller than the threshold 0.2, the action is chosen at random. After selecting an action,

the action set is computed. Finally, every tenth learning step, the XCS checks if rules can be subsumed, i.e., if rules exist that are more general than others and therefore are able to overlap them, the less general rules are deleted and the numerosity of the general rules increased. Additionally, a rule my only act as a subsumption candidate if it has an error value below ε_0 and an experience value above $\vartheta_{subsume}$. These parameters depend on the considered scenario, e.g., the size of the problem space, the number of learning steps in an experiment, and the average number of rule matches.

The second function, *receiveReward()*, takes the reward generated by HALadapt and updates the XCS fitness values (s. Sec. 8.2.4 for the remaining update formulas). In this thesis, the learning rate β and the discount rate γ are set to 0.2 and 0.7, values that are also used in XCSLib. The parameter α is again taken from XCSLib and set to 0.1.

HALadapt's reward mechanism is explained in detail in the following section. As the targeted problem usually is a multi-step problem (s. Sec. 8.3), up to five past rewards are stored to update the action sequence. Additionally, the time difference between two successive task graphs is evaluated. If the time difference is greater than a threshold of ten seconds, the start of a new sequence is presumed and all stored past action sets are updated.

8.5.2 Reward Function

In order for an XCS to be able to learn and improve its rule base and decision making, it needs some form of feedback from the underlying system in response to the selected actions. Generally, two machine learning concepts are associated with XCSs, supervised learning and reinforcement learning (s. Sec. 8.2.3). The use case of the XCS in this thesis is task scheduling, in particular multi-objective task scheduling. Finding the optimal solution for a task scheduling problem is considered NP-hard. Additionally, multi-objective optimization problems usually do not posses a single optimal solution, instead a set of optimal solutions, called the Pareto set (s. Sec. 8.2.1) exists. Supervised learning, however, requires labeled data, i.e., in the case of this thesis, a weighting that leads to an optimal task scheduling. As finding an optimal solution is too computationally expensive for an online feedback mechanism, reinforcement learning is selected as the learning concept for the XCS in this thesis.

Section 8.5 introduced the MLOC framework by Müller-Schloer and Tomforde [9] that consists of an online and an offline layer. Hence, two reward functions for both the online layer that utilizes the XCS and the offline layer that uses an optimization heuristic to generate new rules are needed. As the offline layer allows for more computational overhead, HALadapt's task execution simulator is employed to determine the execution costs of several task schedules created with a variety of different optimization goal weightings. In particular, three optimization goals, makespan, energy consumption, and temperature, are considered in this thesis. Thereby, the goals are to minimize the overall makespan,

to minimize the overall energy consumption, and to minimize the maximum processing unit temperature. These goals can be formalized as follows:

$$\text{Minimize } \max_{1 \leq i \leq N} ft(t_i), \tag{8.6}$$

where t_i is the i-th of N tasks, and $ft(t_i)$ the finish time of task t_i.

$$\text{Minimize } \sum_{1 \leq i \leq N} e(t_i), \tag{8.7}$$

where $e(t_i)$ is the energy needed to execute task t_i.

$$\text{Minimize } \max_{1 \leq j \leq M} T_j(t) \; \forall t \in [0, t_{end}], \tag{8.8}$$

where $T_j(t)$ is the temperature of processing unit j of M system processing units at time t.

The task execution simulator utilizes the profiling data stored in HALadapt's database (s. Sec. 3.3.1 and Sec. 6.2) to compute the task costs. However, the energy costs of an application or task set are not solely determined by the tasks that are executed as idle processing units also consume energy. Especially the introduction of thread variation for OpenMP tasks in Sec. 6.2 may lead to idle slots on certain CPU cores in a task schedule. Therefore, the idle times of the system's processing units have to be considered as well. Thereby, the amount of energy a CPU core or the GPU consumes while being idle was determined empirically by conducting several series of measurements. For the CPU utilized in the evaluation, power values of $1\,W$ and $4.5\,W$ were measured for a single core being idle without and with a CUDA kernel executing in parallel, respectively. The power value measured for the GPU is $56.62\,W$.

To compute the maximum temperature of the processing units during an application or task set execution in the simulator, a lumped RC model [217] is utilized. The model is a simplified version of the HotSpot algorithm by Skadron et al. [218, 219]. Fig. 8.5 shows the processor thermal model. Included in the abstract RC model are a single thermal resistance and a single thermal capacitance, denoted as R and C in the figure, respectively. Additionally, the power consumption P of the chip at time t and the ambient temperature T_{amb} are depicted. Together, this results in the die temperature T. In the model utilized by Zhang and Chatha [217], the temperature T after time t is computed as follows assuming the power consumption P remains constant in the interval $[0, t]$:

$$T(t) = P \cdot R + T_{amb} - (P \cdot R + T_{amb} - T(0)) \cdot e^{\frac{-t}{RC}}, \tag{8.9}$$

where R is the thermal resistance, C the thermal capacitance, T_{amb} the ambient temperature, and $T(0)$ the temperature at time 0. The required values for the power consumption P, the thermal resistance R, and the thermal capacitance C for the CPU and GPU

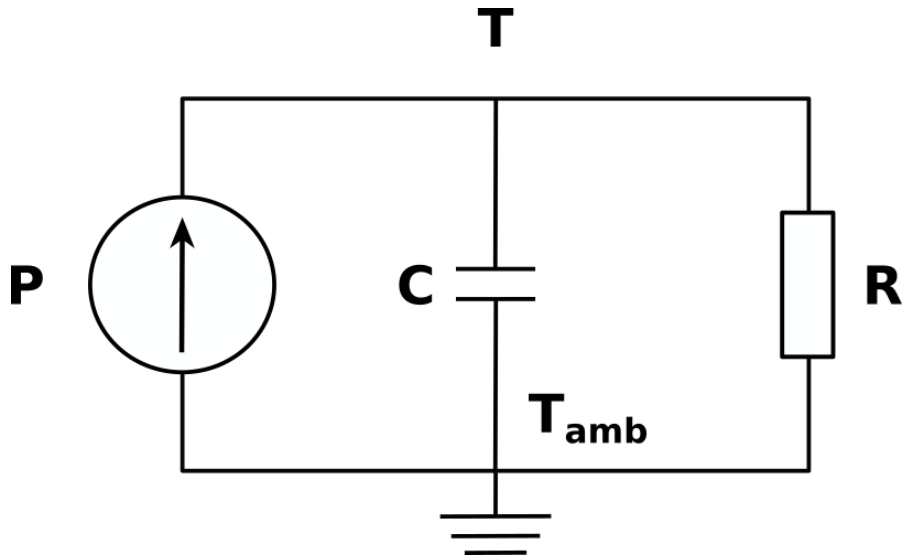


Figure 8.5: Abstract thermal model for the processing units' temperatures [217]

were again determined empirically with several series of measurements. Depending on the core number, the frequency, and the task execution time, the values for the CPU for P, R, and C lie in the ranges of $17\,W$ to $73.5\,W$, $0.27\,\frac{K}{W}$ to $0.49\,\frac{K}{W}$, and $3\,\frac{J}{K}$ to $560\,\frac{J}{K}$, respectively. For the GPU, the values for P and C lie within the ranges of $102\,W$ to $112\,W$, and $80\,\frac{J}{K}$ to $200\,\frac{J}{K}$. For R the single value $0.32\,\frac{K}{W}$ was determined.

For the offline reward function, several task schedules are created with a weighting step size of 0.1 utilizing a multi-objective implementation of the HEFT algorithm [220, 221]. The resulting schedules are compared against each other to find schedules that are not dominated by other schedules. Furthermore, it is checked if the schedules violate potential system constraints. All weightings that lead to schedules that fulfill the constrains and are not dominated by other schedules are suitable actions for the new rule.

An important objective of this chapter is to minimize the response time of the scheduling algorithm. Hence, it is not feasible to create and evaluate several different task schedules to compute a reward value for the online system. So, the schedule created with the selected weight is only compared to a schedule computed with balanced weights. In the case of this thesis that focuses on the optimization objectives makespan, energy consumption, and maximum processing unit temperatures, a balanced weight vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. If the selected schedule dominates the balanced one, a reward of 10 is returned. The other way round, if the balanced schedule dominates, a reward of -10 is returned. If no schedule dominates, 0 is returned.

8.6 Evaluation

In this section, the experimental setup and the evaluation results are presented and discussed. The evaluation was conducted on a server with an Intel Xeon E5-2650 v4 CPU with 12 cores at 2.2 GHz each, an NVIDIA Tesla K80, and 128 GB of 2.4 GHz DDR4 SDRAM DIMM (PC4-19200). The CPUs were utilized without hyperthreading and the scaling governor was set to *performance*. On the server, CentOS Linux 7.9.2009 with the kernel 3.10.0, glibc 2.17, the nvidia-470.103, NVIDIA CUDA 11.4, and the GCC 9.3.1 compiler are installed. All parameters used to parameterize the XCS are shown in Table 8.1. The parameter values have been taken from XCSLib. However, the values for the number of considered past rewards, the number of total rules allowed, δ_{match}, ϑ_{delete}, and $\vartheta_{subsume}$ have been altered to better reflect the properties of the training scenario. The

Table 8.1: Parameter values of the XCS

Parameter	Value
α	0.1
β	0.2
γ	0.7
δ_{delete}	0.1
ε_0	10
ϑ_{delete}	3
$\vartheta_{subsume}$	10
ν	5
δ_{match}	5
Maximum number of rules	400
Maximum number of past rewards	5

evaluation process is separated into two distinct phases. In the first phase, the modified XCS is trained with several runs of training scenarios, i.e., a set of rules is created that serves as the initial rule population for the evaluation phase. A training scenario consists of up to five task graphs forming a pattern that is repeated ten times. Training task graphs are comprised of up to three applications, in particular a multiplication of quadratic integer matrices, particle filter of the Rodinia benchmark suite [123], and a Mandelbrot computation. All applications provide both OpenMP and CUDA implementations for all of their tasks. Furthermore, each OpenMP task may use one of three frequency scaling levels, $1400\,MHz$, $2100\,MHz$, and $2500\,MHz$. Detailed explanations of the applications can be found in Section 8.6.1.

In total, 140 partly randomized patterns are executed in the training phase. A pattern utilized to train the XCS rule base is built as follows:

- A task graph consisting of matrix multiplications with length three and a randomized width between two and five, i.e., between two and five matrix multiplications may execute in parallel and the matrix multiplications are repeated three times on the same input data. So, for a graph width of five, ten input matrices and five output matrices are utilized to create the needed data independence and allow parallel execution. The dimensions of the matrices are randomized between 2000 and 4000 with a step size of 500.
- The second task graph is built with mandelbrot computations. Its length is set to four and the width randomized between three and six. Furthermore, the problem size is randomized between 1000 and 4000 with a step size of 500.
- In every pattern iteration, a task graph with length three combining matrix multiplications and mandelbrot computations is added with a probability of $60\,\%$ after the second graph. Hence, this graph contains three sequential matrix multiplications and three sequential mandelbrot computations. The problem sizes for both applications are taken from the first and second graph, respectively.
- The fourth graph is made up of two to five iterations of particle filter. For all iterations, the same number of particles is utilized. This number is randomized between 10000 and 50000 with a step size of 10000. To every iteration of particle filter, a matrix multiplication and a mandelbrot computation is added with a probability of $70\,\%$ and $50\,\%$, respectively.
- The last graph of the training pattern contains three sequential particle filter execution, three sequential mandelbrot computations, and a single matrix multiplication. The problem sizes for all three applications are taken from the preceding graphs.

In this evaluation, predictions are only simulated and no real model is employed as the prediction component is not yet integrated into HALadapt. Chapter 7, though, proved that it is possible to predict upcoming tasks and applications. As randomly arriving task graphs and tasks cannot be predicted, in this scenario, predictions are only added to the respective system state descriptions between the first and second, and fifth and first task graph. The other graphs are specifically designed with randomized components or are only added to the pattern with a certain probability to simulate aperiodic behavior that cannot be predicted.

After the training phase, an evaluation with a different execution pattern is conducted. The pattern is repeated five times and consists of the following five task graphs:

- The pattern starts with a task graph containing matrix multiplications. The task graph is of length and width three. The parallel multiplications have problem sizes of 2000, 2500, and 3000, respectively.
- For the second task graph, the hotspot3D computation with the problem sizes presented in Sec. 8.6.1 is employed. The graph is initialized with length and width four.
- The third graph is comprised of MLEM and hotspot3D. It computes three iterations of MLEM, i.e., a total of twelve sequential tasks. In parallel to MLEM, the task graph contains four hotspot3D iterations where always two possess a data dependency.
- A fourth graph is added to the pattern in the first, third and fifth pattern repetition. This graph is comprised of matrix multiplications with width two and length three. The problem sizes of the matrix multiplication are 2500 and 3500.
- To build the fifth task graph, all three applications are used. The graph includes three sequential executions of MLEM, three sequential iterations of hotspot3D, and a matrix multiplication with problem size 3000.

Fig. 8.6 shows the first task graph of the pattern used to evaluate the framework. It particularly illustrates that the width y of a graph allows to execute up to y tasks in parallel. The evaluation results are presented in Section 8.6.2.

8.6.1 Applications

Particle filter is a statistical estimator of the locations and paths of target objects in a Bayesian framework given noisy measurements. The benchmark is mainly composed of four distinct kernels, *likelihood()*, *sum()*, *normalizeWeights()*, and *findIndex()*, that are executed in this sequence iteratively over a set of frames. However, in order to combine and interchange the CUDA and OpenMP implementations, the two kernels *likelihood()* and *sum()* had to be combined into a single kernel *likelihoodSum()*. Each kernel is represented by a single task in HALadapt.

Particle filter is defined by four parameters, the width x and height y of a frame, the number of frames z, and the number of particles Np, which have to be set by the user. In the experiments of this chapter, x and y are set to 128. The number of frames z and the number of particles Np are selected randomly between 2 and 4, and 100000 and 500000 with a step size of 100000, respectively. Thereby, the number of frames is equivalent to the number of times the kernels are repeated within a task graph.

Computation of the Mandelbrot set is a parallel processing benchmark that iteratively solves the equation $f_c(z) = z^2 + c$, where $z \in \mathbb{C}$ until the absolute value of the complex number is observed to be either diverging or converging. The maximum number of

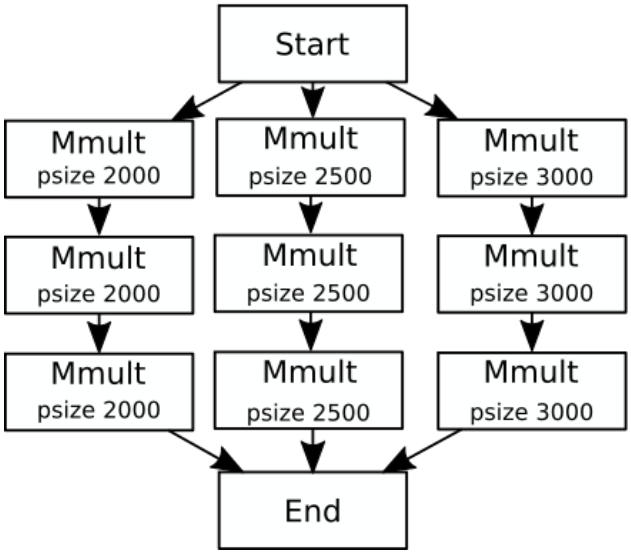


Figure 8.6: Task graph one of the evaluation pattern containing matrix multiplications with width and length three and problem sizes 2000 to 3000

iterations is set to 1000000 and the maximum absolute value to 1024. For visualization, colors are derived by mapping the number of iterations until termination to a color palette, where black indicates convergence. In HALadapt, the computation is represented by a single task with an OpenMP and a CUDA implementation. The Mandelbrot application is defined by a single problem size parameter that is set randomly between 1000 and 4000 with steps of 500. A problem size of 4000 results in the computation of 32000000 complex numbers.

The matrix multiplication implemented for this evaluation multiplies two quadratic integer matrices. Therefore, only a single parameter is needed to define the application, the dimension of a matrix. The dimension is randomized between 2000 and 4000 with a step size of 500.

Hotspot3D iteratively computes the heat distribution of a 3D chip represented by a grid. In every iteration, a new temperature value depending on the last value, the surrounding values, and a power value is computed for each element of the grid. I combined 500 iterations to create a task for HALadapt and provided OpenMP and CUDA implementations for a task. The size of the chip is set to $512 \times 512 \times 8$.

The Maximum Likelihood Expectation Maximization (MLEM) algorithm [222, 223] reconstructs 3D images from data obtained from Positron Emission Tomography (PET) scanners. Reconstructing a 3D image from the scanner readout is an inverse problem. The algorithm requires two inputs: the scanner readout and the system matrix and is therefore defined by their size. Hereby, the system matrix describes the properties of the PET scanner, e.g., the spatial arrangement of the detectors and the physical effects during image acquisition. The system matrix used in this thesis describes the small animal PET scanner Madpet-II [224] and contains 22511722 non-zero elements. It consumes around 173 MB of memory stored in compressed sparse row (CSR) format.

MLEM consists of the following four computation kernels that are all represented by an individual task in its HALadapt implementation:

1. Calculate the estimated scanner readout by using the current image approximation.
2. Calculate a correction vector by correlating the estimated readout with the actual readout.
3. Transfer the correction factor into the image domain.
4. Apply the correction factor and a normalization to the image in order to obtain a new image estimate.

8.6.2 Results

The first goal of the conducted experiments is to evaluate the ability of my approach to improve the balance of the system's optimization objectives and with that the resulting task schedule. Therefore, my approach is compared to the results of a HEFT scheduling algorithm that employs the same weight of $\frac{1}{3}$ to the three optimization objectives makespan, energy consumption, and maximum temperature. The evaluation scenario described in Sec. 8.6 is repeated ten times for both algorithms. After every experiment iteration, the rule set is reset to its state after the training phase. To compare the approaches, the minimum, the maximum, the average, and the standard deviation σ of the overall makespan, energy consumption, maximum GPU temperature, and maximum temperature of a CPU core are monitored and computed. The results are presented in Tab. 8.2. Over the ten experiment iterations, my approach reduces the makespan by $10.4\,\%$ or $26.7\,s$, the energy consumption by $4.7\,\%$ or $2061.1\,J$, and the maximum temperature of the GPU by $3.6\,\%$ or $2.7K$ on average. Only the maximum CPU core temperature is increased by $6\,\%$ or $2.3\,K$. The small values of the standard deviation, e.g., just $0.8\,K$ for the maximum GPU temperature or $2.2\,s$ for the makespan, confirm that these results are stable and are obtained in every iteration of the experiment. Another important research topic is the question if my approach is able to learn rules that

		Makespan	Energy	Max GPU temp.	Max CPU core temp.
XCS & HEFT	min.	$253.6\,s$	$43.5\,kJ$	$74\,°C$	$37\,°C$
	avg.	$257.7\,s$	$43.98\,kJ$	$75.7\,°C$	$38.2\,°C$
	max.	$260.7\,s$	$44.55\,kJ$	$77\,°C$	$39\,°C$
	σ	$2.2\,s$	$283.7\,J$	$0.8\,K$	$0.6\,K$
Balanced HEFT	min.	$284.2\,s$	$44.71\,kJ$	$77\,°C$	$35\,°C$
	avg.	$284.4\,s$	$46.05\,kJ$	$78.4\,°C$	$35.9\,°C$
	max.	$284.6\,s$	$47.65\,kJ$	$79\,°C$	$37\,°C$
	σ	$0.16\,s$	$888.7\,J$	$0.7\,K$	$1\,K$

Table 8.2: Results of the evaluation scenario for the combination of XCS and HEFT and balanced HEFT

are general enough to be utilized outside of their training scenario. In order to answer this question, I monitored the number of rule matches in the first pattern iteration over the ten experiment runs. Only the first iteration is considered because after the first iteration newly generated rules may have been added to the rule set. In all ten experiment iterations, rule matches were registered for the third and fourth task graph. Additionally, in three out of the ten iterations further matches were monitored for task graphs two and five. So, it is fair to say that it is possible to learn generally applicable rules with my approach.

The second objective of this approach is to reduce the response time of the scheduling algorithm while still guaranteeing proactive adaptation and an efficient task schedule. As the generation of new rules can be done offline, the most overhead is produced by the online reward mechanism. To evaluate the overhead, I measured the execution time of the online reward mechanism for all five task graphs of the evaluation scenario. On average, it executed for $0.115\,s$ to compute the necessary reward for the five task graphs. The largest overhead was created for the computation of the second graph with $0.224\,s$. This approximately matches the time HALadapt needs to execute the HEFT algorithm itself. To evaluate the capability of my approach to adapt the system to new situations, an additional experiment is conducted. After two iterations of the evaluation pattern, maximum temperatures for the CPU cores and the GPU are set. These constraints then have to be considered by the XCS. In this experiment, the limit was set to $37\,°C$ for the CPU cores and $69\,°C$ for the GPU. After two iterations of the pattern, the CPU cores have a maximum temperature of $34\,°C$ and the GPU has a maximum temperature of $72\,°C$. For the three remaining pattern iterations, max temperatures of $37\,°C$ and $74\,°C$ were monitored. The average for the maximum temperatures over all applications is $36.1\,°C$ and $73.2\,°C$. Compared to maximum temperatures monitored in the first experiment, this is a decrease for the maximum GPU temperature while keeping the CPU temperature below the set threshold. However, it was not possible to fulfill both constraints as the GPU temperature remains over $69\,°C$.

8.7 Summary and Conclusion

The objective of this chapter is to utilize the knowledge acquired in Chapters 6 and 7 to dynamically and, in particular, proactively adapt the system to avoid disadvantageous system states. This has to be done without the help of exhaustive search algorithms as a fast response time is a necessity for the systems this thesis focuses on. A solution for these challenges is provided by Müller-Schloer's and Tomforde's hierarchical MLOC framework. The framework employs a modified XCS and an offline rule generator to proactively and safely adapt a underlying system. Additionally, the offline rule generation guarantees a fast response time of the XCS.

This chapter proposes an adaptation and implementation of this framework in the context of task scheduling. An important part of an XCS is its reward function that allows the classifier system to evaluate its rules and progress its knowledge. To implement the reward function, this thesis provides a novel task cost simulator that is able to compute the costs of a task graph offline. The simulator employs a simulation model for heat dissipation, the lumped RC model, and utilizes HALadapt's execution database.

To evaluate the framework, the evaluation process was split into two distinct phases. The first phase, the training phase, learns and builds the initial rule set for the evaluation phase. For the training, an execution pattern including randomized elements was deployed (s. Sec. 8.6 for a detailed elucidation of the training pattern). A pattern was repeated ten times and a total of 140 patterns were executed. The evaluation phase uses a pattern of up to five task graphs that is repeated five times. In total, the experiment was repeated ten times. Thereby, my approach reduced makespan by $10.4\,\%$, energy consumption by $4.7\,\%$, and the maximum temperature of the GPU by $3.6\,\%$ while only increasing the maximum CPU core temperature by $6\,\%$. The experiment also showed that my approach is able to learn rules that are applicable outside of their training scenario and that its overhead is in the magnitude of an additional HEFT algorithm execution. Hence, the overhead is usually neglectable compared to the makespan of application patterns and the potential schedule improvements.

To summarize, this chapter provides the control mechanism of this thesis' holistic approach. It utilizes the knowledge acquired in the previous chapters to dynamically and proactively adapt the system to new situations by finding an efficient balance between the system's contradicting optimization objectives. This is achieved without creating significant overhead.

CHAPTER

NINE

TASK-SCHEDULING IN TASK-BASED RUNTIME SYSTEMS

This chapter covers the task scheduling component of the overall approach. The component is used to influence and direct the behavior of the underlying system according to the decisions made in Chapter 8. In particular, the task scheduling component utilizes the weights for the optimization goals provided by the balancing mechanism to evaluate and choose between possible scheduling candidates. In the holistic approach presented in Section 4.2, this chapter is represented by the yellow task scheduling box.

Task scheduling, in general, is a good method to direct the system behavior as it directly determines which task is executed when and where. Thereby, task scheduling controls the system load, the energy consumption, and heat generation of the system. In the context of this thesis, scheduling is strictly limited to mapping tasks to processing units and setting their execution order. This excludes scheduling other resources, e.g., giving exclusive access to a cache to a specific set of tasks. Additionally, preempting tasks that are already executing is also not supported in the scope of this thesis.

In this thesis, two differing scheduling scenarios are considered. The first scenario focuses on dynamically arriving independent tasks of a single process, i.e., all tasks belong to a single runtime system instance. Additionally, priorities are assigned to these tasks to support soft real-time constraints. Here, a dynamic priority adaptation scheme is designed that is able to improve the overall makespan of scheduled applications. The focus of the second scenario lies on the coordination of multiple processes, where each process possesses its own runtime system instance that schedules a set of tasks organized in a directed acylcic graph (DAG). This thesis contributes a co-scheduling mechanism that is able to dynamically re-schedule tasks that are already scheduled but not yet running when a new process arrives in the system. The mechanism, thereby, utilizes shared memory to implement communication between the different processes.

The remainder of this chapter is organized as follows. The formal definition of the scheduling problem and some necessary fundamentals are introduced in Section 9.1. Section 9.2 motivates and discusses the first task scheduling scenario, dynamic scheduling of independent tasks with priorities, and presents my associated scheduling mechanism and its evaluation. Scenario two is the topic of Section 9.3. This includes a motivation of this scenario, the developed scheduling mechanism, and the obtained evaluation results. Finally, the chapter ends by summarizing the two scheduling mechanisms and the insights obtained during their evaluation in Section 9.4.

This chapter contains previously published content [10, 16, 11, 12].

9.1 The Scheduling Problem

The formal scheduling problem is known to be NP-equivalent, which means the associated Entscheidungsproblem is NP-complete [225]. The basic scheduling problem comprises a set of n tasks $T := \{t_1, \ldots, t_n\}$ that has to be assigned to a set of m processing units $P := \{p_1, \ldots, p_m\}$. Next to mapping a task t_i to a processing unit p_j, scheduling also includes the assignment of an ordering and time slices. In the case of heterogeneous systems, the processing units p_j may have different characteristics, which can lead to varying executions times for a single task on different units [220]. The execution times are stored in an $n \times m$ matrix W.

Additionally, a set of tasks can be represented by a directed acyclic graph (DAG) $G = (V,\ E)$, where V is the set of v tasks and E the set of edges between the tasks. An edge $(t_i,\ t_j) \in E$ represents a data dependency between tasks t_i and t_j, which means that task t_i has to be executed before task t_j. The data that has to be moved between the tasks because of the data dependencies are represented by a $v \times v$ matrix D. Data transfer rates between the processors are represented by an $m \times m$ matrix B. So, the communication costs between tasks t_i and t_j run on processors p_l and p_k are defined as $c_{i,j} = \frac{D_{i,j}}{B_{l,k}}$. If two processors use the same memory, the communication costs are 0.

As there is no algorithm that can solve all scheduling problems efficiently, there exist many heuristics. Generally, these can be classified into static and dynamic algorithms [226]. The main difference is that static algorithms make all decisions before a single task is executed, whereas dynamic algorithms schedule tasks at runtime. Hence, static algorithms have to know all relevant task information beforehand, while dynamic ones do not need full information and are able to adapt their behavior.

In this thesis, two different dynamic scenarios are considered. Section 9.2 targets tasks that may potentially create infinite task instances for execution and whose start times may be unknown. This scenario combines minimizing the total makespan with task priorities and hence, considering task starvation and waiting time minimization as optimization goals. In general, task priorities can be set for every instance of a task and

then remain static over its lifecycle, or they are dynamically set for every task instance at runtime and may change over time [184]. The earliest deadline first (EDF) algorithm [227] is a well-known example with dynamic task priorities. Each task instance is assigned the priority $p = \frac{1}{d}$ when it arrives in the system, where d is the deadline of this instance. Contrary to EDF, rate-monotonic scheduling [228] assigns static priorities. Each task is assigned the priority $p = \frac{1}{r_i}$, where r_i is the period of task t_i. In the considered scenario, an application developer is allowed to assign a static priority to a task, which is then used for all instances of this task. However, an *aging* mechanism that is allowed to increase the priority of a single task instance in order to improve fairness if the waiting time of an instance is considered too long is utilized in addition. It has to be noted that task preemption is not supported in this scenario.

Section 9.3 covers the coordination of multiple processes, called co-scheduling. In this scenario, processes with their own set of tasks are started independently and run in parallel. Beforehand it is not known which processes are started and at which point in time a new process arrives in the system.

9.2 Task Scheduling with Priorities

Modern computing systems used in fields like embedded and high performance computing feature a high degree of parallelism and are often equipped with additional accelerators, e.g., GPUs. This parallelism can be used to execute different functionality or applications in parallel as not all applications are able to exploit the available computational power due to a lack of scaling capability. However, executing multiple applications and their corresponding tasks in parallel can be problematic if a certain quality of service is required or expected for a subset of the applications. A common way to express differing importance of applications or functionality in non-safety-critical systems is to assign priorities accordingly. In highly utilized systems though, static task priorities can lead to the starvation of certain tasks. Starvation can be avoided by applying aging mechanisms. Aging refers to the technique of raising the priority of tasks that have waited a certain amount of time in the system for execution. This is not to be confused with hardware aging, where the fault rate of a hardware component increases over its lifetime.

Today's computing systems are often dynamic in nature, which means that the set of tasks to be executed does not remain static, and tasks' start times may be unknown as they may be triggered by signals or user interactions. Therefore, the focus of this section is on dynamic scheduling algorithms. An adaptive aging mechanism that considers the current system state and load is added to avoid starvation.

Generally, heterogeneous architectures present many challenges to application developers. A state-of-the-art solution is offered by task-based runtime systems that abstract from the underlying system and provide helpful functionality for developers. To utilize these features, an existing task-based runtime system, the Embedded Multicore

Building Blocks (EMB^2) (s. Section 3.4), an open-source runtime system and library developed by Siemens is used as a basis to implement the scheduling algorithms. In summary, this section provides the following contributions:

- Six dynamic scheduling algorithms are integrated into a task-based runtime system and the ability to consider task priorities is added to the scheduler module.
- A two-level adaptive aging mechanism is developed to extend the scheduling module.
- The algorithms are evaluated with and without aging on a real system and their behavior in terms of different metrics is investigated.
- The effect of aging in these experiments is analyzed.

In two evaluation scenarios, three independent heterogeneous tasks with differing priorities and sporadic start times for the task instances, and two benchmarks of the Rodinia benchmark suite [123], hotspot3D and particlefilter, in parallel with different priorities, the results show an improvement of total average makespan (average speed up of $3.75\,\%$ and $2.16\,\%$ for five and four out of six algorithms, respectively) while reducing the waiting time of the lower priority tasks.

9.2.1 Related Work

Known existing task-based runtime systems such as HALadapt [60, 5], the TANGO framework [29], and HPX [80] do not employ task priorities to distinguish application importance. StarPU [6], though, supports assigning a priority per processing unit type to a task. Compared to this thesis, StarPU does not adapt priorities at runtime.

Task or job scheduling algorithms with priorities are usually employed in the context of real-time systems, especially hard real-time systems with strict deadlines. These algorithms can be classfied by the way they assign priorities [184]. Algorithms like EDF [227] or least laxity first (LLF) [229] assign each task instance a different priority. Thereby, EDF assigns each instance an individual static priority based on its deadline (s. Sec. 6.1), whereas the priorities assigned by LLF are dynamically adapted as the laxity, the remaining time until a task has to be started to fulfill its deadline, decreases over time [184]. Contrary to this, algorithms like RMS [228] set a static priority that applies to each instance. The work of this paper differs from these algorithms as my tasks do not possess deadlines. In my work, an application developer is allowed to set a priority for a task that then applies to each instance. However, I additionally utilize an aging mechanism to increase fairness, i.e., priorities may be dynamically adapted.

Similarly to EDF, list scheduling algorithms [220, 230, 231] prioritize and then order individual task instances by computing metrics like the upward rank used by the heterogeneous earliest finish time (HEFT) heuristic.

Kim et al. [232] consider task priorities and deadlines in the context of dynamic systems, where the arrival of tasks is unknown. The paper uses three priority levels, high, medium, low, that can be assigned to task instances. The priorities are combined with the tasks' deadlines to compute the worth of executing a task. Thereby, a scheduling order is created. In contrast to our approach, priorities are not dynamically adapted to avoid starvation.

Aging mechanisms have been employed in several other works. Kannan et al. [233] implemented three priority queues and task instances get promoted to a higher priority level after a fixed time interval. Similarly, the priority of a task also gets promoted at fixed time intervalls in [234]. In [235], a counter is decreased after high priority tasks are executed. If a threshold is reached, a low priority task is executed next.

9.2.2 Extensions to EMBB

EMB^2 is designed and implemented in a modular fashion that easily allows developers to add further scheduling policies. However, a few extensions were necessary to implement the selected algorithms and extend them via an aging mechanism.

An abstract scheduling module and a general abstraction for processing units and grouped identical processing units in classes to allow a uniform treatment of all processing units was introduced. Every processing unit is implemented with an OS-level worker thread. Workers corresponding to CPU cores are directly pinned to their respective cores, but are assigned a lower priority than device workers. The idea is that the device worker threads are not used for computation directly, and merely use the CPU for short periods of time to communicate with a device. The task scheduler then submits tasks to the worker threads for execution. Both the task scheduler and the processing unit abstractions use waiting queues to store submitted and assigned tasks, respectively. For EMB^2 to be able to support different task priorities, each component owns a set of queues with one queue for every priority level available in the system. Assigned tasks and tasks ready to execute are then stored in queues according to their current priority.

Scheduling algorithms usually need task execution times to make sophisticated decisions. These can either be given by the user, an analysis step or predicted at runtime. For this algorithm, the focus lies on dynamic systems which means that problem sizes are not known beforehand and static analyses are not possible. Therefore, I extended EMB^2 by a monitoring component that measures task execution times and stores them within a history database with the problem size as key similar to the mechanism used in [5]. As of yet, EMB^2 does not consider data transfers separately. So, a task executed on an accelerator always transfers its data on and off the accelerator regardless of its predecessor and successor tasks. Therefore, the task execution times always include the necessary data transfers. The stored data is then used to predict execution times of upcoming tasks to improve scheduling decisions. If there is already data stored for a

particular task's implementation version and problem size, the data can be used directly. If there is data for a task's implementation version but with different problem sizes, interpolation is utilized to predict the execution time. In the case of no available data, EMB^2 executes this implementation version to create a database entry.

Aging Mechanism

To avoid starvation of tasks, a two-level aging mechanism was added to the scheduling module of EMB^2. The first level is directly part of the scheduler module. Tasks ready to execute are stored into priority-specific ready-queues in the task scheduler. Therefore, if there are n distinct priority levels, n separate ready-queues are created. Each time the scheduler is activated, each non-empty queue with a priority lower than the set maximum priority is checked for potential aging candidates if at least two times the amount of active processing units of tasks are currently ready to execute. So, the aging mechanism is only activated if at least $2 \cdot p$ tasks are enqueued, where p ist the number of currently active processing units. A task in a ready-queue is selected for priority promotion if the task is older than the average task waiting time multiplied with a threshold factor α_{prom}. Formally, this is defined as:

$$\text{promote priority of } t_i \text{ if } waitingTime(t_i) > \frac{\sum_{j=1}^{m} waitingTime(t_j)}{m} \cdot \alpha_{prom}, \tag{9.1}$$

where m is the total number of tasks currently submitted to the task scheduler. After a task is promoted to a new priority queue by increasing its priority, the task is pushed to the back of the queue and its waiting time reset to 0. The second level of the aging mechanism targets the processing units' waiting queues. Each processing unit possesses priority-specific queues, where assigned tasks are stored. Again, if there are n distinct priority levels, each processing unit possesses n separate waiting queues. A task is assigned to the priority level, which it last had in the scheduler. As long as a processing unit is active, i.e., at least one waiting queue is non-empty, each non-empty queue with a priority lower than the set maximum priority is checked for potential aging candidates. Again, a task in a waiting queue is selected for priority promotion if the task is older than the average queue waiting time multiplied with a threshold factor α_{prom} (s. Equation 9.1). Actually, different threshold factors α_{prom} can be used. However in this work, I use $\alpha_{prom} = 1.7$ for both levels. This value was determined empirically as a compromise to reduce overall priority promotion while still enabling the promotion for long-waiting tasks. Again, the waiting time of a task is reset after a promotion and it is pushed to the back of the new queue.

9.2.3 Dynamic Scheduling Algorithms

This section presents the algorithms that have been integrated into EMB^2. The algorithms were selected on the basis of their runtime overhead, since scheduling decisions have to be made as fast as possible in dynamic systems, their implementation complexity, and their ability to work with limited knowledge about the set of tasks to be executed. The selected heuristics can be classified into immediate and batch mode. Immediate mode considers tasks in a fixed order, only moving on to the next task after making a scheduling decision. In contrast, batch mode considers tasks out-of-order and so delays task scheduling decisions as long as possible, thereby increasing the pool of potential tasks to choose from.

Immediate Mode Heuristics

Minimum Completion Time (MCT)
[236] combines the execution time of a task t_i with the estimated completion time ct of the already assigned tasks of a processing unit p_j. In total, MCT predicts the completion time ct of a task t_i and assigns t_i to the processing unit p_j that minimizes ct of t_i.

Batch Mode Heuristics

Min-Min
[237] extends the idea of MCT by considering the complete set of currently ready-to-execute tasks. The heuristic then assigns the task t_i that has the earliest completion time to the processing unit p_j that minimizes the completion time of t_i $ct(t_i)$. In general, the core idea is to schedule shorter tasks first to encumber the system for as short a time as possible. This can lead to starvation of larger tasks if steadily new shorter tasks arrive in the system.

Max-Min
[232] is a variant of Min-Min and based on the observation that Min-Min often leads to large tasks getting postponed to the end of an execution cycle, needlessly increasing the total makespan because the remaining tasks are too coarse-granular to partition equally. So, Max-Min schedules the tasks with the latest minimum completion time first, leaving small tasks to pad out any load imbalance in the end. However, this can lead to starvation of small tasks if steadily new longer tasks arrive.

RASA
[238] is a combination of both Min-Min and Max-Min. It uses them alternatively for each iteration, starting with Min-Min if the number of resources is odd, and Max-Min otherwise. This way, RASA tries to negate the disadvantages of Min-Min and Max-Min respectively and to combine their strengths.

Sufferage
[232] ranks all tasks ready-to-execute according to their urgency based on how much time the task stands to lose if it does not get mapped to its preferred resource. The ranking is given by the difference between the task's minimum completion time and the minimum completion time the task would achieve if the fastest processing unit would not be available. Tasks that do not have a clear preference for a processing unit are prone to starvation.

Relative Cost (RC)
[239] uses the new metric rc, which divides the ct of a task t_i by its average ct over all processing units, to rank tasks. RC both utilizes a static and a dynamic variant of the relative cost metric to compute the final metric. The static variant is defined as $\gamma_s(t_i, p_j) = \frac{et(t_i,p_j)}{et_{avg}(t_i)}$, where $et(t_i, p_j)$ is the execution time of task t_i on processing unit p_j, and $et_{avg}(t_i)$ is the average execution time of t_i over all processing units. $\gamma_d(t_i, p_j)$, the dynamic variant is defined as $\gamma_d(t_i, p_j) = \frac{ct(t_i,p_j)}{ct_{avg}(t_i)}$, where $ct(t_i, p_j)$ is the completion time of t_i on p_j, and $ct_{avg}(t_i)$ is the average ct of t_i over all processing units. The second variant is dynamic as ct is updated after each time a task is mapped to a processing unit. The variants are then combined into $rc = \gamma_s(t_i, p_j)^\alpha \cdot \gamma_d(t_i, p_j)$, where $\alpha \in [0, 1]$ determines the effect of the static costs. In this work, I use $\alpha = 0.5$. RC then maps the task with minimum rc to p_j that minimizes $ct(t_i)$.

9.2.4 Evaluation

As benchmarks, I considered two different scenarios with all benchmark tasks providing both a CPU and a GPU OpenCL implementation. The first scenario consists of three independent heterogeneous tasks with differing priorities and has already been used in previous work [10]. This benchmark resembles dynamic systems as the task instances are started sporadically, thereby adding a random component to the starting point of a task instance.

For the second scenario, two benchmarks of the Rodinia benchmark suite [123], Hotspot3D and Particlefilter, are executed in parallel with different priorities. Both benchmarks distribute their work over several parallel tasks.

All experiments were conducted ten times with and without aging. For each experiment, I measured the makespan of each application or job, and the total makespan

of all tasks. Then, the average, the minimum, and the maximum were computed. The makespan is defined as the time from start to finish of an application or task. Additionally, I measured the flow time of each task and again computed the average, the minimum, and the maximum. The flow time of t_i is defined as $t_{i,\textit{flow}} = t_{i,\textit{finish}} - t_{i,\textit{release}}$, where $t_{i,\textit{release}}$ is the release time or system arrival time of t_i and $t_{i,\textit{finish}}$ is the finish time of t_i. So, $t_{i,\textit{flow}}$ is basically the time t_i spends in the system. It has to be noted that the flow time is usually dominated by a task's waiting time. This potentially leads to large differences between minimum, average, and maximum values.

The experiments were performed on a server with two Intel Xeon E5-2650 v4 CPUs with 12 cores at 2.2 GHz each and dynamic voltage and frequency scaling enabled, an NVIDIA Tesla K80, and 128 GB of 2.4 GHz DDR4 SDRAM DIMM (PC4-19200). The software environment includes Ubuntu 18.04.3, the Linux 4.15.0-74.84-generic kernel, glibc 2.27, and the nvidia-410.48 driver. EMB^2 was compiled with the GCC 7.4.0 compiler. Additionally, EMB^2 was limited to 16 CPU cores for the experiments in order to increase the system load and simulate a highly utilized system.

The scheduling algorithms presented in Section 9.2.3 operate in the so-called pull mode. In pull mode, the scheduler gets triggered iff at least one processing unit is idle. This mode was chosen because it allows the scheduler to accumulate a set of tasks before making a scheduling decision, which is needed to be able to benefit from the batch mode heuristics.

Independent Heterogeneous Jobs

The first scenario is comprised of three video-processing tasks that have both an OpenCL and a CPU implementation:

- J_1 **(Mean)**: A 3×3 box blur.
- J_2 **(Cartoonify)**: Performs a Sobel operator with a threshold selecting black pixels for edge regions and discretized RGB values for the interior. The Sobel operator consists of two convolutions with different 3×3 kernels followed by the computation of an Euclidean norm.
- J_3 **(Black-and-White (BW))**: A simple filter which replaces (R,G,B) values with their greyscale version $(\frac{R+G+B}{3}, \frac{R+G+B}{3}, \frac{R+G+B}{3})$.

All operations were applied to the *kodim23.png* test image. The three operations execute for $72.8\,ms$, $165.97\,ms$, and $11.4\,ms$ on the CPU and $3.4\,ms$, $3.1\,ms$, and $3.1\,ms$ on the GPU. Different priorities were assigned to the three tasks. Mean was assigned the priority 1, Cartoonify the priority 2, and Black-and-White the priority 0 with 2 being the highest and maximum priority in the system. A sporadic profile was used to create instances of these three tasks. New task instances were released with a minimum interarrival time

of $\frac{1}{k}\ secs$, where k is the parameter to control the load, plus a random delay drawn from an exponential distribution with parameter $\lambda = k$. By varying k, I can generate a range of different loads to create different profiles for the scheduling heuristics evaluation. The evaluation workload consists of 3000 task instances corresponding in equal proportions to instances of all three tasks. To simulate a heavily utilized system, the experiment was conducted with $k = 2000$. The results of the makespan measurements can be seen in Table 9.1. They show that for five out of six algorithms the average total makespan is

		MCT	Min-Min	Max-Min	Suff	RASA	RC
Cartoonify	min w/o aging	1.43 s	1.46 s	1.45 s	1.46 s	1.42 s	1.44 s
	min w/ aging	1.68 s	1.64 s	1.79 s	1.51 s	1.79 s	1.59 s
	avg w/o aging	1.49 s	1.56 s	1.54 s	1.53 s	1.59 s	1.52 s
	avg w/aging	1.88 s	1.87 s	2.22 s	1.68 s	2.25 s	1.82 s
	max w/o aging	1.64 s	1.70 s	1.65 s	1.74 s	1.78 s	1.65 s
	max w/ aging	2.38 s	2.24 s	3.67 s	2.23 s	3.71 s	2.69 s
Mean	min w/o aging	2.17 s	2.31 s	2.35 s	2.36 s	2.29 s	2.35 s
	min w/ aging	2.20 s	2.69 s	2.46 s	2.48 s	2.43 s	2.70 s
	avg w/o aging	2.29 s	2.45 s	2.49 s	2.48 s	2.51 s	2.46 s
	avg w/ aging	2.37 s	2.86 s	2.71 s	2.60 s	2.62 s	2.85 s
	max w/o aging	2.55 s	2.82 s	2.70 s	2.86 s	2.93 s	2.70 s
	max w/ aging	2.57 s	3.08 s	3.23 s	2.68 s	3.14 s	3.06 s
BW	min w/o aging	2.38 s	2.75 s	2.79 s	2.83 s	2.68 s	2.85 s
	min w/ aging	2.28 s	2.42 s	2.72 s	2.52 s	2.35 s	2.56 s
	avg w/o aging	2.51 s	2.92 s	2.98 s	2.95 s	2.96 s	2.97 s
	avg w/ aging	2.46 s	2.57 s	2.98 s	2.74 s	2.58 s	2.80 s
	max w/o aging	2.77 s	3.27 s	3.23 s	3.32 s	3.38 s	3.27 s
	max w/ aging	2.70 s	2.74 s	3.23 s	2.84 s	2.75 s	2.99 s
Total	min w/o aging	2.38 s	2.75 s	2.79 s	2.83 s	2.68 s	2.85 s
	min w/ aging	2.31 s	2.69 s	2.72 s	2.62 s	2.52 s	2.70 s
	avg w/o aging	2.51 s	2.92 s	2.98 s	2.95 s	2.96 s	2.97 s
	avg w/ aging	2.47 s	2.86 s	3.03 s	2.74 s	2.82 s	2.90 s
	max w/o aging	2.77 s	3.27 s	3.23 s	3.32 s	3.38 s	3.27 s
	max w/ aging	2.70 s	3.08 s	3.67 s	2.82 s	3.71 s	3.06 s

Table 9.1: Makespan results of the independent heterogeneous jobs experiment

improved by adding the aging mechanism, with Max-Min being the only algorithm where the makespan increases by $1.6\,\%$ or $0.05\,s$. On average over all algorithms that show an improvement, the average makespan is improved by about $3.75\,\%$. Sufferage profits the most with an improvement of about $7.5\,\%$ or $0.24\,s$. Considering the single applications, aging increases the average makespan for Cartoonify by about $26.9\,\%$ or $0.42\,s$ and for Mean by about $8.9\,\%$ or $0.22\,s$ compared to a decrease of $6.7\,\%$ or $0.19\,s$ for Black-and-White. Especially for Max-Min and RASA, which uses Max-Min, the average makespan of Cartoonify suffers from an increase of over $40\,\%$. Other noteworthy results are an increase of over $13.7\,\%$ for the maximum measured total makespan for Max-Min and of over $9.5\,\%$ for RASA, which correlates with an increase of $123.1\,\%$ and $108\,\%$ respectively for Cartoonify. Comparing the algorithms, MCT achieves the best average total makespan with and without aging while Max-Min achieves the worst result in both cases. Sufferage gets the second best results in both cases.

Further, I obtained results for the flow time $t_{i,flow}$ of each task instance t_i and then computed the minimum, average, and maximum flow time for all three tasks. Table 9.2 lists the results. The results show a significant increase, by $95.2\,\%$ or $0.15\,ms$ on av-

		MCT	Min-Min	Max-Min	Suff	RASA	RC
Cartoonify	min w/o aging	1.54 ms	1.61 ms	1.46 ms	1.49 ms	1.46 ms	1.48 ms
	min w/ aging	1.77 ms	2.00 ms	1.47 ms	1.56 ms	1.51 ms	1.61 ms
	avg w/o aging	214.01 ms	239.59 ms	222.93 ms	229.04 ms	233.67 ms	220.88 ms
	avg w/ aging	477.43 ms	355.65 ms	427.44 ms	230.28 ms	730.11 ms	202.69 ms
	max w/o aging	1047.54 ms	948.90 ms	1500.34 ms	1137.52 ms	1000.74 ms	1124.99 ms
	max w/ aging	1306.71 ms	1553.85 ms	3624.45 ms	1454.86 ms	3313.17 ms	1838.82 ms
Mean	min w/o aging	1.81 ms	1.72 ms	1.62 ms	1.72 ms	1.64 ms	1.64 ms
	min w/ aging	2.99 ms	6.05 ms	1.65 ms	1.81 ms	1.66 ms	1.79 ms
	avg w/o aging	1073.45 ms	1208.23 ms	1206.62 ms	1261.46 ms	1145.06 ms	1226.23 ms
	avg w/ aging	904.87 ms	1092.57 ms	1030.84 ms	1073.86 ms	886.34 ms	1213.63 ms
	max w/o aging	1498.08 ms	2570.64 ms	2708.11 ms	2513.19 ms	2180.18 ms	2605.41 ms
	max w/ aging	1459.64 ms	26617.62 ms	3196.53 ms	1904.88 ms	2830.09 ms	2883.13 ms
BW	min w/o aging	1.41 ms	1.69 ms	1.33 ms	1.41 ms	1.44 ms	1.57 ms
	min w/ aging	1.76 ms	1.76 ms	1.49 ms	1.37 ms	1.48 ms	1.73 ms
	avg w/o aging	1744.71 ms	2085.71 ms	2085.25 ms	2142.52 ms	2016.48 ms	2123.33 ms
	avg w/ aging	1400.28 ms	1333.99 ms	1946.73 ms	1587.36 ms	1263.95 ms	1517.70 ms
	max w/o aging	2365.10 ms	2823.32 ms	3116.59 ms	2914.91 ms	2817.72 ms	3026.06 ms
	max w/ aging	2089.93 ms	2161.86 ms	3210.39 ms	2321.71 ms	2402.84 ms	2666.41 ms

Table 9.2: Flow time results of the independent heterogeneous jobs experiment

erage, in the average flow time for Cartoonify in 5 out of 6 experiments, with RC being the exception. For Cartoonify, this correlates with an increase in the maximum flow time for each algorithm. In contrast, the average flow time for both Mean and Black-and-White decreases for each algorithm by $13.1\,\%$ or $153.2\,ms$ on average and $25.67\,\%$ or $524,7\,ms$ on average, respectively. This shows the desired effect of the aging mechanism as the waiting time for instances of both tasks is reduced by increasing their priority if the waiting time has become too long. For Black-and-White, this also correlates with a decrease in the maximum flow time measured.

Parallel Applications

The second scenario consists of two Rodinia benchmark applications, hotspot3D and particle filter, executed in parallel. Particle filter was assigned the priority 1, and hotspot3D the priority 0.

Hotspot3D iteratively computes the heat distribution of a 3D chip represented by a grid. In every iteration, a new temperature value for each grid element depending on the temperature value of this element in the last iteration, the surrounding values, and a power value is computed. The computation was parallelized over the z-axis by integrating it into EMB^2. The CPU implementation then further splits its task into smaller CPU specific subtasks. This is done manually and statically by the programmer to use the underlying

		MCT	Min-Min	Max-Min	Suff	RASA	RC
Particle filter	min w/o aging	26.46 s	26.52 s	25.97 s	26.57 s	26.62 s	25.61 s
	min w/ aging	27.25 s	25.96 s	26.37 s	26.82 s	26.23 s	26.42 s
	avg w/o aging	27.61 s	27.73 s	27.82 s	27.72 s	27.56 s	27.85 s
	avg w/ aging	27.76 s	27.17 s	27.49 s	27.61 s	27.62 s	27.67 s
	max w/o aging	28.79 s	27.92 s	28.83 s	28.54 s	28.79 s	29.56 s
	max w/ aging	28.62 s	28.37 s	28.55 s	28.97 s	29.15 s	29.47 s
Hotspot3D	min w/o aging	26.84 s	27.82 s	29.91 s	30.52 s	26.63 s	30.22 s
	min w/ aging	26.27 s	25.37 s	26.02 s	25.93 s	25.33 s	27.82 s
	avg w/o aging	30.93 s	30.59 s	31.44 s	31.38 s	30.71 s	31.78 s
	avg w/ aging	30.70 s	30.59 s	30.03 s	30.96 s	30.64 s	30.93 s
	max w/o aging	32.60 s	32.18 s	32.02 s	32.28 s	32.09 s	33.46 s
	max w/ aging	31.81 s	31.91 s	31.71 s	32.47 s	31.83 s	33.16 s
total	min w/o aging	26.84 s	27.82 s	29.91 s	30.52 s	26.63 s	30.22 s
	min w/ aging	27.42 s	26.68 s	26.62 s	26.82 s	26.48 s	27.82 s
	avg w/o aging	30.93 s	30.59 s	31.44 s	31.38 s	30.71 s	31.78 s
	avg w/ aging	30.81 s	30.72 s	30.09 s	31.05 s	30.76 s	30.93 s
	max w/o aging	32.60 s	32.18 s	32.02 s	32.28 s	32.09 s	33.46 s
	max w/ aging	31.81 s	31.91 s	31.71 s	32.47 s	31.83 s	33.16 s

Table 9.3: Makespan results of the Rodinia benchmarks experiment

parallelism of the multicore CPU and still have a single original CPU task that handles the same workload as the GPU task. For the evaluation, a $512 \times 512 \times 8$ grid with the temperature and power start values that are included in the benchmark, and 1000 iterations were utilized as parameters. The average runtime on the CPU is $5.03\,ms$ and $7.36\,ms$ on the GPU.

Particle filter is a statistical estimator of the locations of target objects given noisy measurements of that target's location and an idea of the object's path in a Bayesian framework. Profiling showed that *findIndex()* is the best candidate for a parallelization as it consumes the most execution time. *findIndex()* computes the first index in the cumulative distribution function array with a value greater than or equal to a given value. As *findIndex()* is called for every particle, the computation was parallelized by dividing the particles into work groups. The CPU implementation again further divides those groups into subtasks. I employed the standard parameters 128 for both matrix dimensions, 100 for the number of frames, and 50000 for the number of particles for the evaluation. The average task runtime on the CPU is $17.8\,ms$ and $6.5\,ms$ on the GPU. Table 9.3 shows the makespan results for this experiment with and without aging, respectively.

The average total makespan is improved for four out of six algorithms when aging is used, with Min-Min and RASA being the exceptions. Over these four algorithms, the average total makespan is improved by a speed up of about $2.16\,\%$ or a decrease of $0.66\,s$. In this scenario, Max-Min profits the most from using the aging mechanism with a speed up of about $4.5\,\%$ or a decrease in average total makespan of $1.35\,s$. The individual average makespans decrease by $0.6\,\%$ for particle filter and by $1.6\,\%$ for Hotspot3D over

		MCT	Min-Min	Max-Min	Suff	RASA	RC
Particle filter	min w/o aging	4.58 ms	4.56 ms	4.52 ms	4.92 ms	4.59 ms	4.34 ms
	min w/ aging	4.29 ms	4.37 ms	4.68 ms	4.65 ms	4.18 ms	4.45 ms
	avg w/o aging	42.16 ms	42.10 ms	43.93 ms	43.61 ms	42.19 ms	42.21 ms
	avg w/ aging	43.35 ms	41.98 ms	43.46 ms	41.69 ms	42.66 ms	43.47 ms
	max w/o aging	564.30 ms	568.94 ms	606.79 ms	663.52 ms	528.12 ms	723.96 ms
	max w/ aging	636.67 ms	506.50 ms	490.95 ms	639.28 ms	651.32 ms	543.53 ms
Hotspot3D	min w/o aging	2.34 ms	1.96 ms	2.15 ms	1.92 ms	1.94 ms	1.67 ms
	min w/ aging	1.63 ms	1.65 ms	1.62 ms	1.63 ms	2.33 ms	1.62 ms
	avg w/o aging	13.33 ms	13.06 ms	13.27 ms	13.16 ms	13.25 ms	13.49 ms
	avg w/ aging	13.31 ms	13.28 ms	13.16 ms	13.31 ms	13.17 ms	13.37 ms
	max w/o aging	932.72 ms	801.60 ms	838.39 ms	686.93 ms	648.79 ms	819.58 ms
	max w/ aging	709.43 ms	674.13 ms	656.37 ms	938.87 ms	664.43 ms	735.07 ms

Table 9.4: Flow time results of the Rodinia benchmarks experiment

all scheduling heuristics. It is also noteworthy that the minimum obtained makespan of hotspot3D decreases by over $13\,\%$ for both Max-Min and Sufferage. Comparing the scheduling heuristics, Min-Min achieves the best average total makespan without aging and Max-Min the best result with aging, with Min-Min getting the second best result.

Again, we additionally monitored the flow time $t_{i,\mathit{flow}}$ for all task instances t_i and computed the minimum, average, and maximum over all instances for both applications. The results are shown in Table 9.4.

The results show a decrease in the minimum and maximum flow time of hotspot3D for 5 and 4 algorithms, respectively. This correlates with shorter waiting times caused by a priority raise. The averages roughly remain unchanged. This can be explained by the much larger number of tasks for hotspot3D, which results in many hotspot3D tasks being executed after particle filter is finished. Hence, these tasks that do not need and profit from aging dominate the average for hotspot3D, which means the average remains roughly unchanged.

9.2.5 Result Discussion

The results show a slight improvement in total average makespan (average speed up of $3.75\,\%$ and $2.16\,\%$) for 5 out of 6 algorithms in the first and for 4 out of 6 algorithms in the second scenario. As expected, this correlates with an increase in average makespan for the applications with higher priorities caused by additional waiting time (the total time spent in queues in the scheduler and processing unit). This is also reflected in the flowtime measurements. The average increase of $95.2\,\%$ for the average flowtime of cartoonify is exemplary for this statement. However, the average flowtime and the average makespan of the application/task with the highest priority remain lowest over all applications/tasks in all experiments. Thereby, the intention of the user to favor these applications is preserved. In return, the aging mechanism reduces the waiting time, which is reflected by improvements of the average makespan and the average flowtime, of the

task/application with the lowest priority ($25\,\%$ decrease in average flowtime for black-and-white). A comparison between the scheduling algorithms shows that no algorithm dominates the other ones considering the average total makespan. MCT, Sufferage, and RC, though, are able to profit in all experiments by using aging.

In summary, our adaptive aging mechanism slightly improves the overall makespan in most experiments while reducing the time a low priority task instance has to wait for its execution, thereby increasing fairness, and still securing the fastest execution and shortest time spent in the system for the task with the highest priority. In the future, supplemental evaluations are necessary to further solidify these conclusions. Furthermore, additional optimization goals next to fairness and makespan, like energy consumption, have to be considered in future work.

9.3 Scheduling Multiple Processes

Modern computer architectures combine a steadily growing number of heterogeneous processing units in a single system to satisfy the ever increasing performance demand. Many existing applications are, however, not ready for such an increased degree of intra-node parallelism as their scaling capabilities are limited. This prevents efficient system usage, potentially leaving several computing resources idle at runtime. Prior work proposed an approach called *Co-Scheduling* to combat this problem. Co-Scheduling aims to schedule multiple, different applications on the same computing node simultaneously. This approach works best, if these applications have complimentary characteristics, e.g., one application is memory-bound and one compute bound or one application targets an accelerator and one the CPU. Additionally, multiple applications usually do not belong to the same process requiring a mechanism that coordinates different processes on one computing node. Since new processes with new tasks may arrive aperiodically in the system, the co-scheduling has to be adapted dynamically to improve the current system state.

This section presents a solution for the aforementioned challenges by expanding HA-Ladapt's shared memory waiting queue concept presented in Section 3.3. Thereby, additional data structures in shared memory are introduced that are able to share information about the tasks to be executed with other processes. This information is then employed by an added co-scheduling mechanism The mechanism is additionally able to redistribute tasks already mapped to a processing unit, however does not support preemption.

In particular, this section makes the following contributions:

- Data structures that store information about tasks to be executed, e.g., execution costs, task dependencies etc., are added in shared memory to share with other HALadapt instances.

- A co-scheduling mechanism that uses the information stored in shared memory to optimize the distribution of tasks belonging to different processes is developed. The mechanism also considers tasks already mapped to processing units that have not started execution.
- Different strategies that decide when the computation of a new co-schedule should be initiated are implemented.

Using several use case scenarios including Rodinia Benchmark [123] applications and a Mandelbrot computation, to evaluate this approach, I show that my approach is able to achieve a speedup of atleast $17.7\,\%$ compared to an approach that does not adapt its decisions if new tasks arrive in the system.

9.3.1 Related Work

Various studies in the literature shed light on the many contexts where co-scheduling is applicable. They thereby make up numerous hot topics in recent research, due to the fact that the necessity of computational co-scheduling arises wherever independent jobs in separate processes are scheduled in parallel to multiple execution devices. Therefore, co-scheduling occurs on different levels of abstraction.

Besides work focusing on the co-scheduling of processes in a virtual machine environment [240, 241], a large field of research is made up of co-scheduling jobs on homogeneous architectures, i.e., symmetric multicore processors (SMPs). There, effort is spent on examining approaches for co-scheduling [242, 243, 244], finding (approximately [245]) optimal co-schedules [246], or regard multi SMP systems [247]. The presented approaches mostly focus on solving issues regarding cache contention, which is not necessarily a predominant concern within the field of heterogeneous computing.

To support the co-scheduling of multiple processes, inter-process communication is required. Standards like the Message Passing Interface (MPI) [3] enable such mechanisms. StarPU [6, 248], a runtime system for heterogeneous systems (s. Sec. 3.5 for more details) also supports MPI and offers scheduling contexts that allow to share a GPU between several kernels or divide a multicore CPU into mulitple sections. However, these contexts have to be defined by the application developer and are not dynamic. Additionally, there is no communication between multiple StarPU instances and therefore only a single application workload is scheduled. Similarly, the runtime system Uintah [249, 250], specifically designed for three-dimensional grid simulations, utilizes MPI to communicate over several computing nodes. However, again only a single application workload can be scheduled and communication is restricted to a single MPI application.

The runtime system Dandelion [251] distributes sequential code over a heterogeneous cluster by assigning graphs to machines and graph vertices to specific processing units. The data synchronization is implemented by asynchronous channels. Compared

to this work, Dandelion only distributes a single workload over multiple processing units and does not share processing units between multiple workloads.

The LAMA framework [252] also enables inter-process communication, but focuses mainly on mathematical applications and thereby introduces some limitations with respect to the user's application diversity.

Newsom et al. implemented a co-scheduling mechanism for CPU clusters that optimizes energy usage based on simulation results [253]. The scheduler is written in Unified Parallel C (UPC) and schedules MPI applications on a SMP cluster and adjusts system parameters like the CPU frequency to reduce energy consumption. The mechanism of this thesis additionally considers heterogeneous accelerators like GPUs during its co-scheduling process.

Inter-process communication based on shared memory is employed by schedGPU [254] to schedule multiple CUDA applications on a single GPU. Here, shared memory is used to communicate the state of the GPU's memory in order to ensure that parallelly running CUDA kernels do not run out of memory and solely kernels that all can satisfy their memory requests are executed parallelly. Scheduling policies like first in, first out and maximum memory utilization are utilized to decide which waiting kernel is allowed to allocate newly available memory. In contrast to this work, the co-scheduling is limited to GPUs.

Jiménez et al. [255] also utilize shared memory to share scheduling information between multiple processes. In their work, Jiménez et al. introduce three scheduling algorithm concepts. The most sophisticated algorithm is based on a performance history. For every task, a set of allowed processing units is created that includes the processing units without a big performance unbalance. The algorithm then tries to map each task to a processing unit of the set. If all units are currently busy, the scheduler selects the processing unit that minimizes the waiting time of the task. Compared to this work, the scheduling algorithms do not alter their scheduling decisions when new processes or tasks arrive in the system.

9.3.2 Scheduling Algorithms Background

This section introduces and elucidates the scheduling algorithms, the HEFT algorithm and simulated annealing (SA), that are utilized in the co-scheduling mechanism of this thesis. Particularly, the parametrization of SA is discussed in detail as these parameters determine the efficiency and potency of the algorithm.

Heterogeneous Earliest Finish Time Algorithm

The Heterogenous Earliest Finish Time (HEFT) algorithm [220, 221] is a popular list scheduling heuristic for heterogeneous systems. HEFT sorts all tasks by computing a

specific priority called upward rank. The upward rank u of a task i depends on its average execution and communication costs and is defined as follows:

$$\mathrm{rank}_u(i) = \overline{\mathrm{W}_i} + \max_{j \in \mathrm{succ}(i)} (\overline{\mathrm{c}_{i,j}} + \mathrm{rank}_u(j)), \tag{9.2}$$

where $\mathrm{succ}(i)$ is the set of i's successors. Tasks are sorted in a list in decreasing order of the upward rank and then, in this order, mapped to the processing units. Each task is mapped to the processing unit that minimizes its finish time.

Simulated Annealing

Simulated Annealing (SA) is an iterative search based optimization heuristic [256] and therefore offers a way to overcome local minima, a feature a greedy heuristic does not posses. SA is based on the annealing process of molten materials [257], where in every iteration the temperature is decreased. Precisely, it is based on the observation that a slow and controlled annealing process encourages the creation of especially uniform crystals, which resembles an optimization process because a uniform crystal is the energetic minimal structure.

In the scheduling context the Simulated Annealing algorithm usually is used as depicted in Algorithm 1.

SA includes several parameters and functions that are defined as follows:

- $S_{initial}, S_{new}, S_{best}, S$ represent the initial, new, best, and last accepted schedule.
- $C_{initial}, C_{new}, C_{best}, C$ are the costs of the initial, new, best, and last accepted schedule.
- T_{start}, T_{end}, T represent the starting, final, and current temperature of the algorithm.
- R_{max}, R are defined as the maximum and current number of sequentially rejected new schedules.
- L represents the number of algorithm iterations per temperature level.
- $\mathrm{create\_new\_schedule}(S)$ computes a new schedule by modifying the last accepted schedule.
- $\mathrm{accept\_new\_schedule}(\Delta C, T)$ depending on the current temperature, decides if a new schedule is accepted.
- $\mathrm{compute\_new\_temperature}(i, T_{start}, L)$ computes the temperature for the next algorithm iteration.

Some of these parameters and functions have to be specified by the algorithm developer as there exists a wide range of different possibilities in the literature.

Algorithm 1 Simulated Annealing

Constants: $S_{initial},\ C_{initial}\ , T_{start},\ T_{end},\ R_{max},\ L$
Temp variables: $S \leftarrow S_{initial},\ S_{best},\ C \leftarrow C_{initial},\ C_{best},\ T,\ R,\ i$
while ! terminated(T, T_f, R, R_{max}) **do**
  for $l \leftarrow 1$ to L **do**
    $S_{new} \leftarrow$ create_new_schedule(S)
    $C_{new} \leftarrow$ evaluate_schedule(S_{new})
    $\Delta C \leftarrow C_{new} - C$
    if $\Delta C \leq 0$ or accept_new_schedule$(\Delta C, T)$ **then**
      $(S,\ C) \leftarrow (S_{new},\ C_{new})$
      $R \leftarrow 0$
      if $C_{new} < C_{best}$ **then**
        $(S_{best}, C_{best}) \leftarrow (S_{new},\ C_{new})$
      end if
    else
      $R \leftarrow R + 1$
    end if
  end for
  $i \leftarrow i + 1$
  compute_new_temperature(i, T_{start}, L)
end while
return S_{best}

Parametrization of SA

The SA parameters start and end temperature T_{start}, T_{end}, the maximum number of sequential rejections R_{max}, and the temperature level L all heavily influence the algorithm runtime and hence have to be chosen carefully. In [258] an extensive study of these parameters was done by Orsila et al. For T_{start} and T_{end}, we selected the following definitions based on Orsila et al.:

$$\begin{aligned} T_{start} &= \frac{t_{max}}{t_{minsum}} \\ T_{end} &= \frac{t_{min}}{t_{maxsum}}, \end{aligned} \tag{9.3}$$

where t_{max} is the longest execution time of any task on any processor, t_{min} is the fastest execution time of any task on any processor, t_{minsum} is the minimal sum of execution times on one processor, which means the sum of execution times on the on average fastest processor, and t_{maxsum} is the maximum sum of execution times on one processor, which means the sum of execution times on the on average slowest processor.

The definitions of R_{max} and L are also inspired by Orsila et al. However, instead of using the system's number of processing units as a factor, I utilize the number of processing unit groups as the number of processing units has grown substantially in the last years. The definition is as follows:

$$L = R_{max} = N \cdot (Q_{class} - 1), \tag{9.4}$$

where N is the number of tasks and Q_{class} is the number of processing unit groups in the system. I already evaluated these parameters in past work [259]. To compute a new temperature in each iteration, I chose the standard method "geometric temperature schedule" $T_i = T_0 * q^{\lfloor \frac{i}{L} \rfloor}$, where $q = 0.95$.

Finally, computing a new schedule is an essential step in the algorithm. First, a global task order based on an initial solution by HEFT is created. To compute a new schedule, a task is randomly selected and a new position in the global order is determined. The new position is computed by searching the task predecessor with the highest and the successor with the lowest index in the global task order. Then, the task is randomly placed in between the computed interval in the global order and all other tasks are reordered accordingly. In the final step, an implementation variant and a mapping to processing units for the selected task is chosen randomly.

9.3.3 Shared Memory Data Structures

To implement further coordination between multiple HALadapt processes, the runtime system's use of the shared memory is extended. Suitable data structures that hold information on tasks of all currently existing HALadapt instances in the system as well as

flags for signaling purposes are added. In detail, I further introduce two different data structures, a central management file and task specific information files, which were necessary as pointer dependencies make it impossible to store the complete task struct used by HALadapt in shared memory:

- Co-Scheduling Information Management File (CSIMF): This file may exist only once per computing node. Its purpose is to hold information that processes need for coordination and signaling of a co-scheduling procedure and allows processes for a quick check whether further action is required. The header of the CSIMF consists of mutexes, the number of CSI files that processes have stored, the co-scheduling's status and the process ID of the initiator thereof. The header is followed by a variable amount of CSIMF entries, of which every entry corresponds to a CSI file that a HALadapt instance has stored in the shared memory. The CSIMF entries hold various task information that a scheduler can access before looking up the actual CSI file which can be useful for scheduling algorithms. This information includes the current mapping of the task, i.e., on which processing unit(s) a task should be executed, and an algorithm-specific task priority.
- Co-Scheduling Information (CSI) File: The CSI files hold task specific information needed by a scheduler for decision making. Its header contains information on execution dependencies of the respective task, and its current best index, i.e., the position of the currently best fitting execution alternative in the payload according to the current state of the scheduling. The CSI file's aforementioned payload is of variable length and holds information on execution times of the specific task implementations. A CSI file also possesses a trailer that contains information about the input and output data of a task to enable the computation of necessary data transfers and their cost. The sheer presence of a CSI file in the shared memory implies that the corresponding task is not running yet, rendering it subject for a co-scheduling. Processes delete their tasks' CSI files as soon as the respective task is started.

9.3.4 Co-Scheduling Mechanism

Figure 9.1 gives an overview of the overall control flow of the co-scheduling mechanism. The scheduling process of a HALadapt instance begins by assembling the tasks submitted by the application into a task graph and computing a schedule with a "local" scheduling algorithm. Additionally, the tasks are stored in shared memory to provide information to potential other HALadapt processes. After a schedule is computed, HALadapt checks if a co-schedule is necessary considering the new system state created by the task allocations. Thereby, I implemented several different strategies that a user can pick to decide if a co-schedule should be computed:

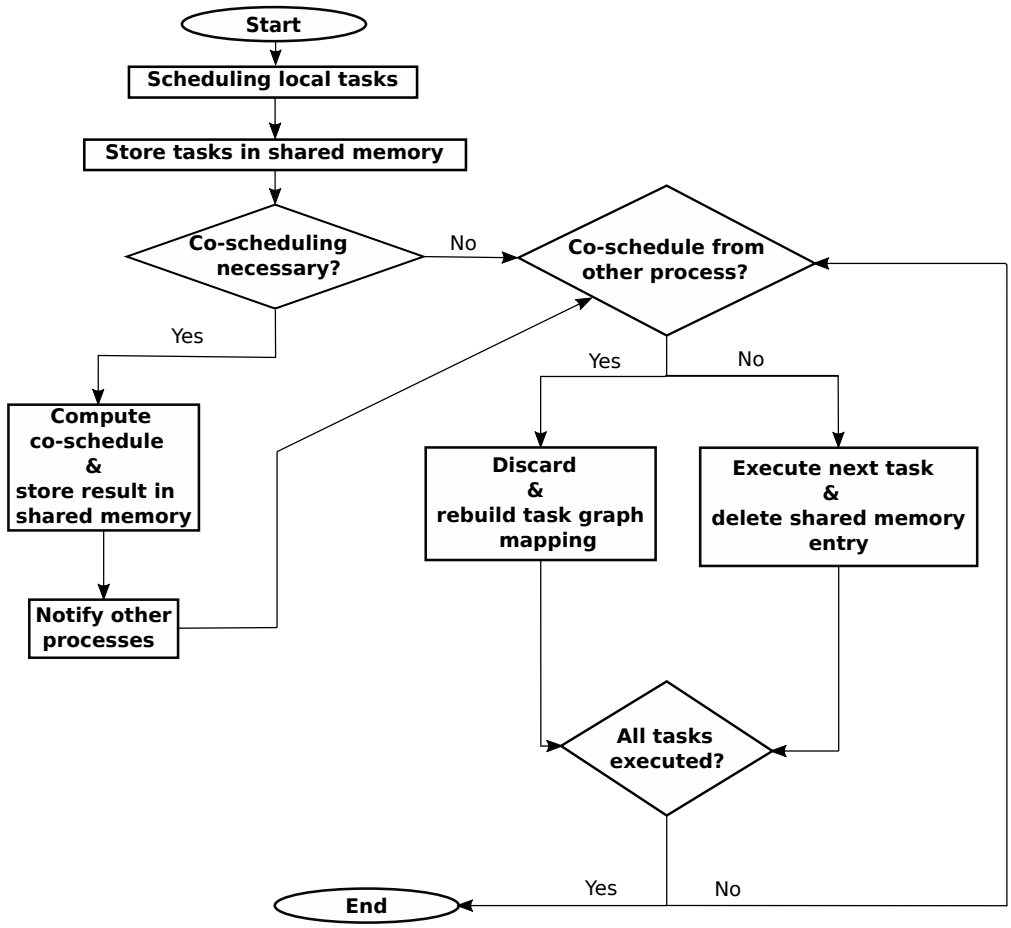


Figure 9.1: General overview of the control flow of the co-scheduling mechanism

- Always: each time a HALadapt instance adds new tasks to the system, a new global schedule is computed.
- Hardware contention: a new schedule is deemed necessary, if a new task is mapped to a processing unit that is already occupied by tasks of another process.
- Processing unit availability: a new schedule is only computed, if the processing units are occupied for at least the amount of time it takes to compute the new schedule. The costs to compute a new schedule are heuristically predicted by the history database. The profiling mechanism monitors the execution costs of the scheduling process and stores the measurements with the sum of implementation variants over all tasks as key. This allows to predict future scheduling costs.

If based on the selected strategy the computation of a new co-schedule is deemed necessary, the shared memory data structures are locked, signaling the computation of a new schedule. The mechanism then collects all tasks and task information stored in the shared memory, creates an initial schedule utilizing HEFT (see Sec. 9.3.2) and computes all parameters necessary for SA (see Sec. 9.3.2). The initial schedule serves as starting point for SA. As mentioned before, it is not possible to store the original task structs of HALadapt in shared memory. Therefore, local processing units were added that can work with the task information stored in shared memory and are used to compute processing unit availability. Following the creation of a final schedule solution by SA, the result is stored in shared memory, and additionally a flag is set and the shared memory structures are unlocked to signal all other processes that a new schedule is available. Subsequently, every HALadapt instance maps its actual tasks according to the stored schedule and starts its execution. Before each execution, our mechanism checks if another process has computed a new global schedule and re-maps its tasks if necessary until all its tasks are executed.

9.3.5 Evaluation

This section presents the evaluation conducted for the co-scheduling mechanism. The experiments were performed on a server with two Intel Xeon E5-2650 v4 CPUs with 12 cores at 2.2 GHz each, an NVIDIA Tesla K80, and 128 GB of 2.4 GHz DDR4 SDRAM DIMM (PC4-19200). The CPUs were used without hyper-threading and the scaling governors set to *performance*. The software environment includes Ubuntu 18.04.5, the Linux 4.15.0-136-generic kernel, glibc 2.27, the nvidia-410.48 driver, and the GCC 7.5.0 compiler. For the evaluation, the co-scheduling mechanism is set to the mode *hardware contention* (s. Section 9.3.4).

In the experiments, two scenarios are considered. The first scenario targets OpenMP applications that compete for the CPU cores. Thereby, a scaling application is executed in parallel with a memory-bound application. In the second scenario, the focus lies on applications than can be executed either on the CPU or the GPU, i.e., have OpenMP and CUDA implementations. As evaluation metric, the total makespan over all applications is measured. In this case, this includes all initialization costs including the set up of HALadapt and all data structures, the scheduling overhead, and the execution costs. All measurements are repeated ten times and the average, maximum, and minimum are computed over these measurements.

Benchmarks

To evaluate my co-scheduling mechanism, I created two scenarios, a CPU-only and a heterogeneous scenario, that are comprised of three experiments using a set of benchmarks that represent common workloads in parallel computing. The benchmark set con-

sists of four benchmarks, hotspot3D and particle filter of the Rodinia Benchmark Suite [123], a Mandelbrot computation, and an example implementation of an OpenMP kernel that does not scale with a growing number of threads. I integrated these benchmarks into HALadapt, thereby dividing the applications into tasks. The following sections shortly give an overview of the benchmarks and how they are divided into tasks for HALadapt.

Hotspot3D iteratively computes the heat distribution of a 3D chip represented by a grid. In every iteration, a new temperature value depending on the last value, the surrounding values, and a power value is computed for each element of the grid. I combined 500 iterations to create a task for HALadapt and provided OpenMP and CUDA implementations for a task. As input arguments, I use the power and temperature data provided by Rodinia in the dimension $512 \times 512 \times 8$. Fig. 9.2 shows the average execution time of a single task execution of the hotspot3D OpenMP kernel, i.e., 500 iterations, over all available thread numbers. Each measurement was repeated ten times. The graph

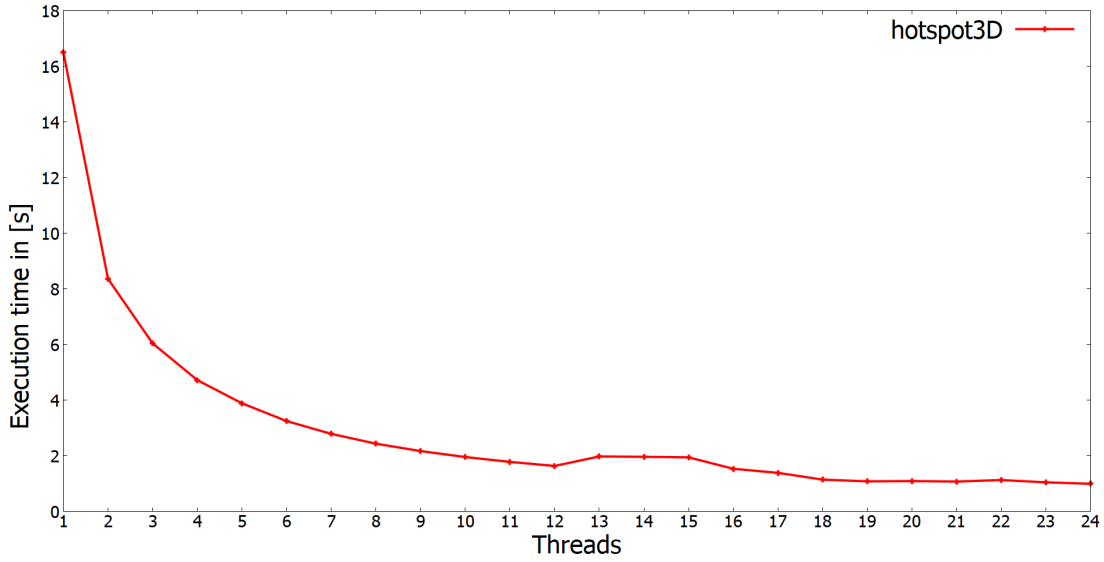


Figure 9.2: Average execution times of the hotspot3D kernel

demonstrates that hotspot3D scales well with the growing number of threads and the lowest average execution time is achieved with 24 threads. Hereby, the lowest average execution time is $0.995\,s$. Additionally, a small spike in the execution times at and after 13 threads is visible. This can be explained with the fact that a single processor consists of twelve cores and splitting the computation and data over two cores creates overhead if only a small number of threads operates on the second CPU. The CUDA kernel has an average execution time of $0.155\,s$ on the NVIDIA Tesla K80 of the test server.

Particle filter is a statistical estimator of the locations and paths of target objects in a Bayesian framework given noisy measurements. The benchmark is mainly composed of four distinct kernels, *likelihood()*, *sum()*, *normalizeWeights()*, and *findIndex()*, that are executed in this sequence iteratively over a set of frames. However, in order to combine and interchange the CUDA and OpenMP implementations, the two kernels likelihood and sum had to be combined. I used each of these three kernel functions to create tasks and provided both an OpenMP and a CUDA implementation for each kernel. For the evaluation, the grid parameters 128×128, five frames, and 100000 particles were utilized. Thereby, the three kernels compute each frame sequentially, i.e., the three kernels are repeated five times each. The average execution times of the particle filter OpenMP kernels for a single frame are displayed in Fig. 9.3 and Fig. 9.4. Again, each measurement was repeated ten times. Hereby, findIndex() clearly dominates the execution time of particle filter. FindIndex() scales well with the growing number of threads and

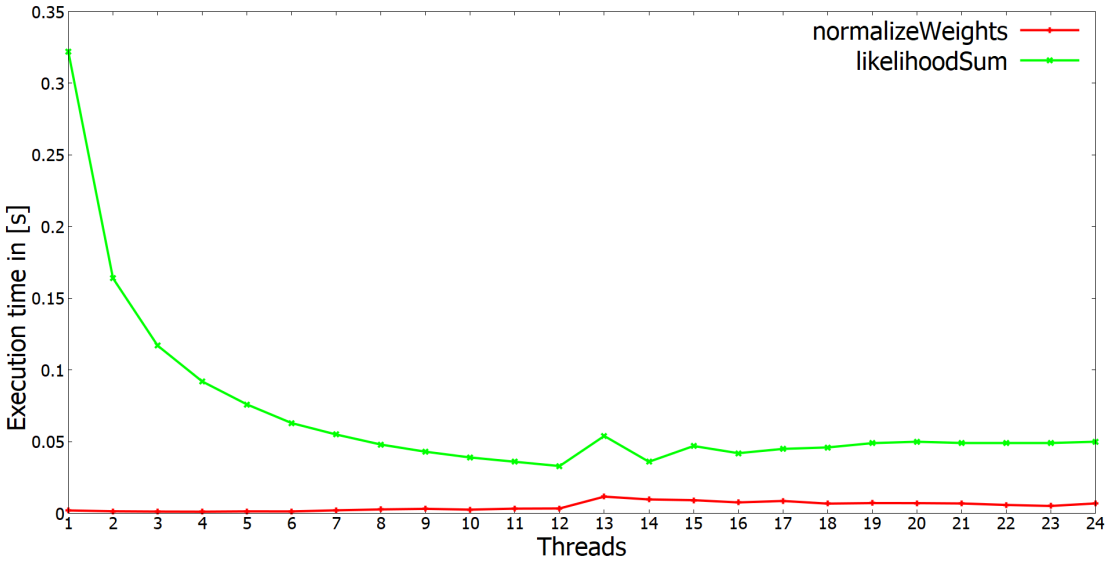


Figure 9.3: Average execution times of the normalizeWeights and likelihoodSum kernels

reaches its minimum average execution time of $1.347\,s$ with 24 threads. LikelihoodSum() also scales with a growing number of threads, however its minimum average execution of $0.033\,s$ time is achieved with 12 threads. As normalizeWeights() almost has no computational costs, the kernel does not scale well and the minimum average execution time of $0.00119\,s$ is already achieved with $four$ threads. Any additional threads only produce overhead and thereby increase the execution time. Again, all three kernels show the same spike symptom at and after 13 threads. On the Tesla K80, the CUDA implementations of likelihoodSum(), normalizeWeights(), and findIndex() have an average execution time of $0.099\,s$, $0.033\,s$, and $0.102\,s$ over ten measurements, respectively.

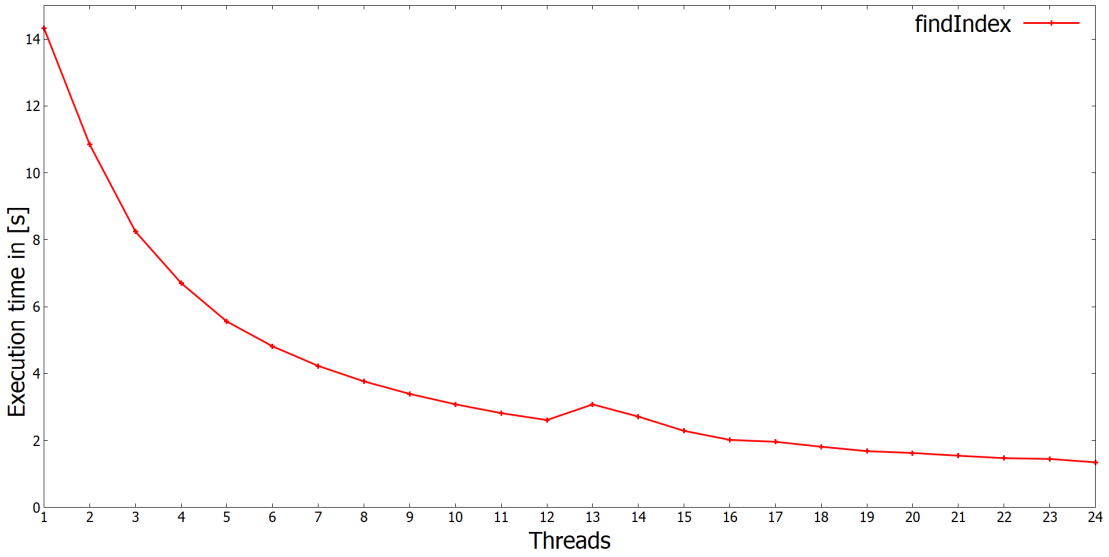


Figure 9.4: Average execution times of the findIndex kernel

Computation of the Mandelbrot set is a parallel processing benchmark that iteratively solves the equation $f_c(z) = z^2 + c$, where $z \in \mathbb{C}$ until the absolute value of the complex number is observed to be either diverging or converging. For visualization, colors are derived by mapping the number of iterations until termination to a color palette, where black indicates convergence. I integrated the Mandelbrot set computation as a single task into HALadapt and again, provided both an OpenMP and a CUDA implementation. In the experiments, two different problem sizes, a large and a small workload, are utilized. The small workload computes 2000000 complex numbers, the large one 18000000 complex numbers. The Mandelbrot benchmark is only used for the heterogeneous co-scheduling experiment. Therefore, only the average execution times of the GPU kernel and the OpenMP kernel with all 24 cores are stated here. With 24 threads, the OpenMP kernel has an average execution time of $0.455\,s$ for the small workload and $3.9282\,s$ for the large one. The GPU kernel achieves an average execution time of $0.145\,s$ for the small workload and $0.443\,s$ for the large one on the Tesla K80.

Critical section kernel is a representation test implementation of an OpenMP kernel that does not scale well with a growing number of threads because its parallelism is hindered by a critical section and random memory accesses. The kernel executes math operations like additions, logarithms, powers, and trigonometric functions, on integers over $2,000,000$ iterations. An overview of the average execution time over ten measurements of the kernel is presented in Fig. 9.5. As intended, the kernel does not scale with

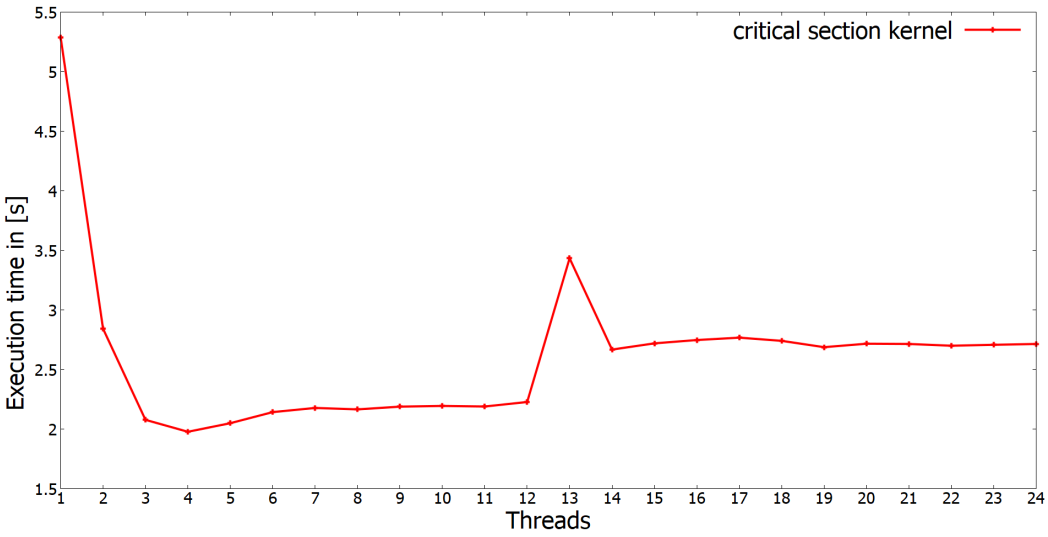


Figure 9.5: Average execution times of the critical section kernel

the growing number of threads and has its minimum average execution time at $four$ threads with $1.978\,s$. The kernel also shows the same symptom as the other kernels when the split over the two processors occurs.

CPU Co-Scheduling

In this scenario, two OpenMP applications are executed in parallel and have to compete for the CPU cores. Thereby, a schedule without co-scheduling is compared to a schedule of the co-scheduling mechanism.

The first experiment of the OpenMP scenario combines hotspot3D with the critical section kernel. Both applications are integrated into HALadapt and are started with their own separate HALadapt instance. Hotspot3D is started 1.5 seconds before the critical section kernel and both kernels are repeated five times. Thereby, the kernels of each application are data dependent, i.e., kernel two of hotspot3D uses the output of kernel one as input data and so on. This scenario is executed with and without the co-scheduling mechanism enabled and all measurements are repeated ten times.

The experiments without the co-scheduling mechanism resulted in the following task schedule. As hotspot3D is started first and is scaling well on the test system (s. Fig. 9.2), HALadapt maps the five hotspot3D kernels consecutively to all 24 available cores. When the second application critical section kernel is started, all cores are already occupied. Therefore, all five critical section kernels are scheduled consecutively after all hotspot3D kernels are finished. Here, HALadapt maps the kernel to $four$ cores as the kernel achieves its minimum execution time with $four$ threads. In total, an average

makspan of $16.114\,s$ was achieved with this schedule. The result is also listed in Table 9.5 with additional information in the form of the minimum and maximum makespan over all ten measurements. With the co-scheduling mechanism enabled, HALadapt creates

Table 9.5: Total makespans of the hotspot3D and critical section scenario

	W/o co-scheduling	With co-scheduling
Average makespan	$16.114\,s$	$13.482\,s$
Speedup	–	$19.52\,\%$
Minimum makespan	$15.725\,s$	$13.055\,s$
Maximum makespan	$16.747\,s$	$14.092\,s$

the following schedule. First, all five hotspot3D kernels are again mapped to all 24 available cores. When the second process is started, the co-scheduling mechanism detects a resource conflict between hotspot3D and the critical section kernel. As the first of five hotspot3D kernels already executes, it is not interrupted. However, for the remaining four kernels and all five critical section kernel instances, a new schedule is computed by the co-scheduling mechanism. Thereby, the mechanism maps the critical section kernels consecutively to the first four CPU cores. The four hotspot3D kernels are then mapped in parallel to the critical section kernels to the last 20 cores. This means that the co-scheduling mechanism shares the number of cores between the two kernels and executes them in parallel. Figure 9.6 displays the computed co-schedule. The schedule resulted in an average total makespan of $13.482\,s$, a speedup of $19.52\,\%$ or a makespan reduction of $2.632\,s$ compared to the execution without the co-scheduling mechanism (s. Tab. 9.5).

The second OpenMP experiment pairs the critical section application with particle filter. Both applications are repeated five times with data dependencies between each iteration. This means particle filter executes 15 kernels in total as it consists of the three kernels *likelihoodSum()*, *normalizeWeights()*, and *findIndex()*. Again, the critical section kernels are started $1.5\,s$ after particle filter.

Without the co-scheduling mechanism, the particle filter functions are mapped to twelve, four, and 24 cores respectively as these numbers minimize the execution time for likelihoodSum, normalizeWeights, and findIndex, respectively. The critical sections are then mapped to four cores after all particle filter kernels have finished their execution. This is necessary as the execution of an iteration of likelihoodSum and normalizeWeights is not long enough to execute a critical section kernel instance in parallel. Over ten measurements an average total makespan of $17.315\,s$ was achieved with this schedule without co-scheduling. Table 9.6 presents this result and additional information. When the co-scheduling mechanism is activated, it detects the resource conflict at the start of the five critical section kernels. At this point in time, the first three kernels, one

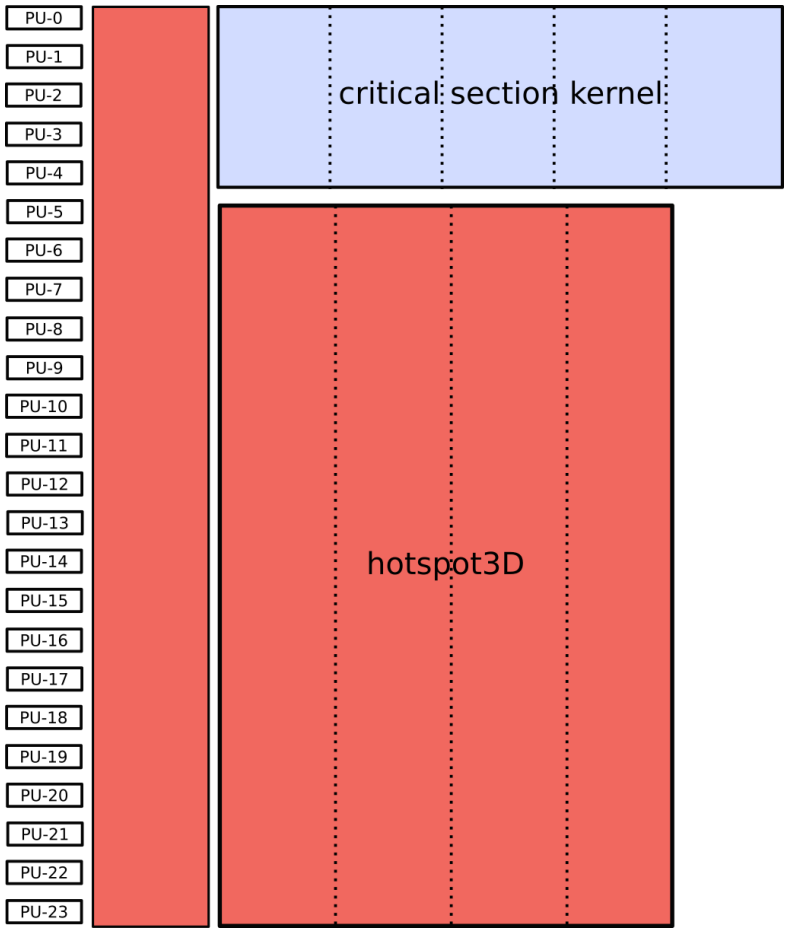


Figure 9.6: The schedule of hotspot3D and the critical section kernel created by the co-scheduling mechanism

instance of each likelihoodSum, normalizeweights, and findIndex, of particle filter have already been executed or are currently executing. All three particle filter kernels have again been mapped to twelve, four, and 24 cores, respectively. The remaining twelve kernels are then re-scheduled together with the five critical section kernels. Thereby, the following schedule is created. The five critical section kernel instances are mapped to the first four cores. All particle filter kernels are scheduled to execute in parallel to the critical section kernels. LikelihoodSum and normalizeWeights are again mapped to twelve and four cores. FindIndex is mapped to the remaining 20 cores that are still unused by the critical section kernel. This schedule results in a average total makespan of $14.321\ s$ and a speedup of $20.91\ \%$ compared to the execution without the co-scheduling mechanism. The results are also presented in Table 9.6 and the computed schedule is displayed in Figure 9.7.

Table 9.6: Total makespans of the particle filter and critical section scenario

	W/o co-scheduling	With co-scheduling
Average makespan	$17.315\,s$	$14.321\,s$
Speedup	–	$20.91\,\%$
Minimum makespan	$16.879\,s$	$13.789\,s$
Maximum makespan	$17.712\,s$	$15.138\,s$

The experiments also showed that executing two kernels in parallel comes with an overhead. If this is not considered in the database of the profiling mechanism, profiling measurements that are conducted with additional kernels running in parallel can lead to suboptimal decisions when compared to measurements without the slowdown of parallel kernels. To solve this problem, measurements with parallelly executing kernels have to be marked and stored separately from "normal" measurements.

Heterogeneous Co-Scheduling

The objective of this section is the co-scheduling of two processes on a heterogeneous platform consisting of a multicore CPU and a GPU. Both processes can be executed on the CPU and the GPU, i.e., have an OpenMP and a CUDA kernel implementation. For the experiment, a scenario that executes two mandelbrot processes, one with a smaller and one with a larger workload, is utilized. The small workload is comprised of 2000000 complex numbers and the large one of 18000000 numbers (s. Sec. 9.3.5). In the experiment, both workloads are each subsequently executed five times. The large workload process is started $0.1\,s$ before the small workload process. However, the small workload kernels still start with their execution as its initialization overhead is smaller and therefore the scheduler maps these kernels first.

Without the co-scheduling mechanism all ten kernels are scheduled in sequence to the Tesla K80 as it minimizes the execution time of the kernels. This leads to an average total makespan of $3.596\,s$. Further details are listed in Tab. 9.7. When the scenario is

Table 9.7: Total makespans of the mandelbrot scenario

	W/o co-scheduling	With co-scheduling
Average makespan	$3.596\,s$	$3.055\,s$
Speedup	–	$17.71\,\%$
Minimum makespan	$3.487\,s$	$2.989\,s$
Maximum makespan	$3.715\,s$	$3.152\,s$

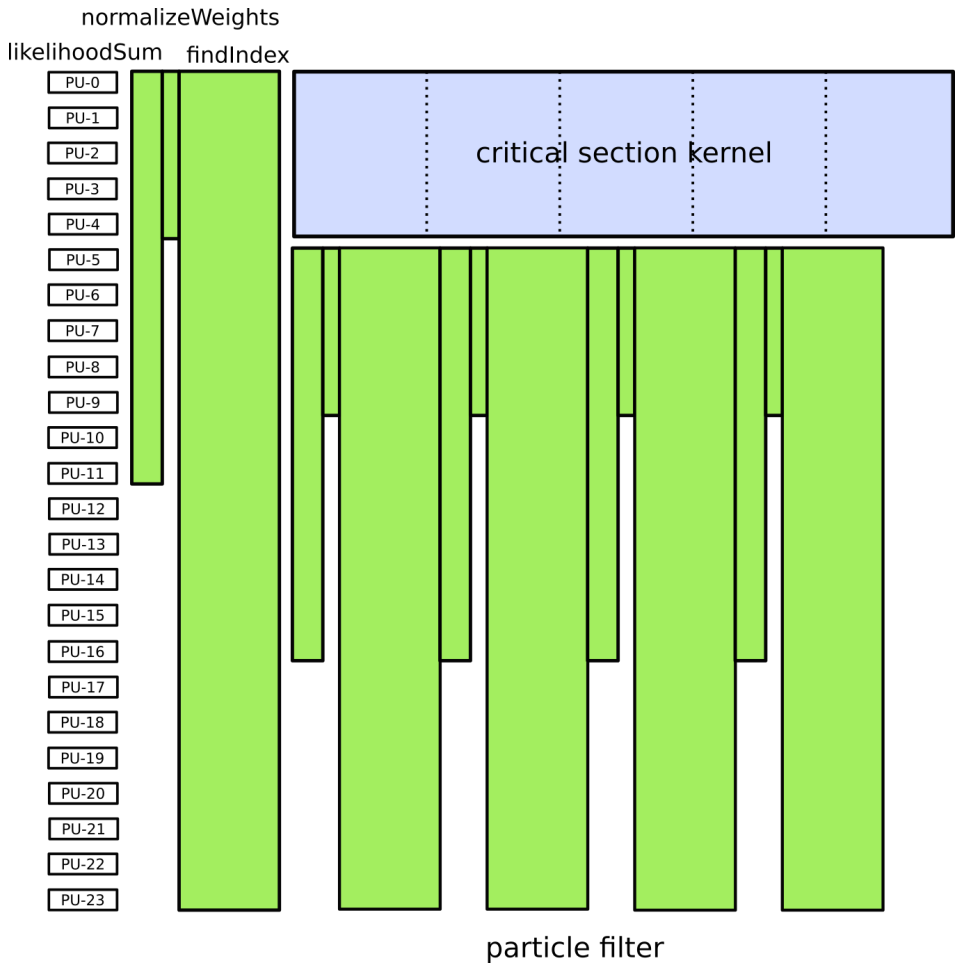


Figure 9.7: The schedule of particle filter and the critical section kernel created by the co-scheduling mechanism

executed with the co-scheduling mechanism, a resource conflict is detected when the mandelbrot kernels with the larger workload (called mandelbrot B from here on forward) are also scheduled to the GPU. The mechanism then re-schedules the remaining four mandelbrot kernels with the lower workload (called mandelbrot A from here on forward) to the CPU using all 24 available cores. In summary, the following schedule is created:

- All five kernels of mandelbrot A are scheduled to the GPU by its HALadapt instance. The first kernel starts its execution.
- The five kernels of mandelbrot B are also scheduled to the GPU by its HALadapt instance. Thereby, the co-scheduling mechanism detects a resource conflict and re-schedules all nine kernels not already running .

- The co-scheduling mechanism schedules the four remaining kernels of mandelbrot A to the 24 available CPU cores. Furthermore, it schedules the five kernels of mandelbrot B to the GPU.

Executing this scenario with the co-scheduling mechanism results in an average total makespan of $3.055\,s$. This is a speedup of $17.71\,\%$ or a makespan reduction of $0.541\,s$ compared to the original schedule.

9.3.6 Result Discussion

The evaluation shows that the co-scheduling mechanism is able to improve the total makespan of the processes in both the CPU-only and the heterogeneous scenario. Over both scenarios, the mechanism could achieve at least a speedup of $17.71\,\%$. This underlines the benefit potential of sharing resources between applications and dynamically adapting schedules to newly arriving tasks and processes. However, the experiments also showed some limitations of this concept.

Executing kernels in parallel comes with an overhead that occurred in both scenarios. Firstly, this overhead has to be considered in HALadapt's profiling mechanism and database. Measurements of kernels that are executing in parallel with other kernels have to be separated from "normal" measurements as mixing them together then falsifies the original measurements and may lead to suboptimal scheduling decisions. Additionally, a scaling factor is needed that adds the overhead to "normal" measurements when the kernel is utilized in a co-scheduling scenario and executed in parallel with another kernel. This is necessary to realistically simulate and predict the results of scheduling decisions, which is a requirement for a sophisticated scheduling mechanism.

Secondly, the overhead that came with sharing computing resources differs between different applications and system settings. So, a speedup in the CPU-only experiments was only possible with hyper-threading deactivated and the scaling governors of the CPU cores set to *performance* instead of *powersave*. Furthermore, no speedup could be achieved when the hotspot3D benchmark was executed on the CPU while parallelly executing the mandelbrot benchmark on the GPU as hotspot3D was considerably slowed down. Therefore, in the future, further analysis and research into the cause of this slow down is necessary. Particularly, application characteristics have to be studied in order to determine which characteristic combinations may cause problems. This will allow to predetermine which applications are ideal candidates for sharing processing units.

9.4 Summary and Conclusion

In this thesis, future system behavior is affected by dynamically adapting the schedule of the system's tasks. Therefore, this chapter focuses on two dynamic task scheduling

scenarios and provides mechanisms that improve the overall makespan of the resulting schedules compared to their static solutions.

The first mechanism targets dynamic systems that allow the application developer to assign specific priorities to different applications or functionalities in order to express differing importance. EMB^2 (s. Sec. 3.4), a task-based runtime system specifically designed for the parallel programming of embedded systems, serves as the basis of the mechanism. As static priorities may easily lead to task starvation, an adaptive aging mechanism was added to the scheduling module. The aging mechanism dynamically adapts the priorities of task instances according to their waiting time and the load in the system. Thereby, the aging mechanism operates on two levels, the scheduler module and the processing units themselves. On both levels, priority waiting queues are implemented that hold the task instances. These queues are checked for potential aging candidates, which are then promoted to the next highest queue. The aging mechanism was combined with six different dynamic scheduling algorithms and evaluated on two scheduling scenarios. The first scenario consists of three independent heterogeneous tasks with differing priorities that are started sporadically to resemble a dynamic system. For the second scenario, two Rodinia benchmark applications, hotspot3D and particle filter, are executed in parallel with differing priorities. The results show a slight improvement in total average makespan (average speed up of $3.75\,\%$ and $2.16\,\%$) for five out of six algorithms in the first and for four out of six algorithms in the second scenario. This is achieved while overall reducing the time a low priority task instance has to wait for its execution, thereby increasing fairness, and still securing the fastest execution and shortest time spent in the system for the task with the highest priority.

With the second mechanism of this chapter, the focus lies on scenarios where multiple processes are run in parallel on a heterogeneous computing node. Hereby, the scheduling mechanism improves the overall makespan of the processes by intelligently sharing processing units and re-adjusting computed schedules to newly arriving processes. In the literature, this process is known as co-scheduling. HALadapt, a task-based runtime system (s. Sec. 3.3) into which this mechanism was integrated, deploys shared memory structures to communicate processing unit utilization between different processes. This communication mechanism is extended by additional shared memory data structures that enable the processes to share task information necessary for re-scheduling. Furthermore, a dynamic co-scheduling mechanism is added that utilizes the information in shared memory to adjust the task schedule if it is activated by the arrival of a new process in the system. The co-scheduling mechanism is evaluated in two scenarios, a CPU-only and a heterogeneous one, and three experiments overall. In all three experiments, the mechanism at least achieves a speedup of $17.71\,\%$ compared to the execution without co-scheduling. However, the experiments also brought limitations to light, e.g., sharing the CPU cores may require some additional settings and not all applications are compatible for sharing processing units.

In summary, this chapter adds mechanisms to the existing scheduling modules of the runtime systems EMB^2 and HALadapt that improve their schedules in dynamic scenarios. This is an important feature of this thesis, as task scheduling in the context of this thesis is utilized to affect the future behavior of the underlying system and to dynamically adapt the system to new situations.

Part IV

Summary and Outlook

CHAPTER

TEN

CONCLUSION AND OUTLOOK

The final chapter of this thesis summarizes its contributions and results, and provides a conclusion. Additionally, this chapter gives an outlook on future work that may sensibly extend this thesis and further enhance its results and contributions.

10.1 Summary & Conclusion

The overarching objective of this thesis is to design and create dynamic and proactive system adaptation within a runtime system for heterogeneous systems. Thereby, the proactive adaptation shall be utilized to maintain efficiency concerning multiple contradicting optimization goals, in particular application makespan, total energy consumption, maximum processing unit temperatures, and system reliability. To achieve this objective, this thesis makes contributions to five research topics (s. Sec. 4.1):

I. Analysis of task-based runtime systems in different use cases and computing classes of heterogeneous systems to find and identify the specific optimization goals, constraints, and requirements.
II. A thorough study of methods and tools to capture the system state by monitoring the execution behavior and the system environment, and the introduction of new possibilities to evaluate the system and reduce profiling overhead. This includes a mechanism to heuristically measure the reliability of processing units.
III. The exploration of ways to augment the system state description by the prediction of prospective system sates, in particular predicting upcoming applications and task instances.
IV. The research of methods to proactively and dynamically balance contradicting optimization goals based on both user inputs and alterations of the system state.
V. Studying ways to influence future system behavior via task scheduling.

Contributions I to III are the focus of Part II of the thesis. In Chapter 5, I analyzed all projects that I have done over the course of this thesis and compared their project objectives and constraints. It became obvious that all projects had several contradicting optimization goals and that the optimal balance between these goals is always dependent on the state of the system and its environment. Therefore, a runtime system is required that is able to dynamically and proactively balance and optimize the underlying system in different situations.

The basis for sophisticated decision making is knowledge. In case of a runtime system, this knowledge is acquired by monitoring the execution behavior of the system. This thesis contributes twofold to the research topic of monitoring system behavior. Firstly, creating an extensive knowledge base requires significant overhead. Hence, the first contribution reduces this overhead through mechanisms that detect scaling behavior, utilizes interpolation to predict task costs, and utilizes machine learning to predict the execution times of OpenCL kernels. However, the results of the kernel execution time prediction showed that a reliable prediction still needs additional work and only the fastest processing unit can reliably be predicted. This, however, is possible with an accuracy of $69\,\%$ in the worst case compared to a baseline of $25\,\%$ that always guesses the processor that dominates the most amount of kernels.

The second contribution is a heuristic metric for the reliability of processing units. For the metric's computation, I employ symptom-based fault detection. Symptom-based fault detection is a light-weight mechanism to detect soft errors in hardware during runtime by monitoring the execution behavior via performance counters and comparing the monitored behavior against a database storing correct execution behavior. If a significant deviation from the correct behavior is monitored, the occurrence of a fault is assumed. This allows to compute a heuristic and dynamic fault rate for processing units, which can be utilized as a measure for their reliability. To evaluate the concept of symptom-based fault detection, several fault classes and faults that simulate hardware faults and interferences were injected into CPU and GPU benchmarks. The evaluation showed that all faults could be detected by symptom-based fault detection on both the CPU and GPU. To analyze the statistical significance of the observed performance metric deviations and to statistically back up the usefulness of symptom-based fault detection, I applied Welch's t-test. Thereby, the t-test results confirmed the impression of the evaluation.

Though, achieving proactivity requires more than just monitoring the current state of the system and its environment. Proactivity also requires knowledge about states in the future or at least in the near future. Therefore, I introduced two task/application prediction mechanisms based on run length encoding (RLE) Markov predictors combined with prediction by partial matching (PPM) in Chapter 7. The first mechanism targets independent tasks that may start new instances of this task during their life cycle. Hence, the idea of the prediction mechanism is trying to detect the execution patterns of the tasks in the system and to differentiate between periodic and aperiodic patterns. My mecha-

nism was able to reliably detect regular patterns and predict the time difference between two instances with a maximum $sMAPE$ value of $4.33\,\%$ for sporadic and $0.002\,\%$ for periodic tasks.

I designed the second predictor to focus on dependent applications, i.e., applications that form execution patterns, e.g., application B always succeeds application A. Again, the task of the mechanism is to detect regular patterns while ignoring interferences from randomly arriving applications. The objective of the mechanism is to predict the next upcoming application and its starting time. Over ten experiments, my mechanism achieved an average accuracy of $77.6\,\%$. Thereby, a prediction was deemed correct if the next application is correctly predicted and its predicted starting time does not deviate more than $0.5\,ms$ or if no prediction is made and the next application is not part of a regular pattern. It is noteworthy, that my mechanism is designed in such a way that false positives, i.e., predicting the wrong application, are strongly avoided which can be seen in the evaluation results where no false positive prediction occurred. However, avoiding false positives by, e.g., requiring a certain amount of pattern occurrences before accepting a pattern, reduces the amount of true positives, i.e., making a correct prediction.

Part III of my thesis provides contributions IV and V. Chapter 8 utilizes the knowledge provided by Part II to dynamically and proactively find a suitable compromise between the system's optimization objectives. This is achieved by a hierarchical Organic Computing framework that includes an XCS to map system state descriptions to a weight vector. To implement this framework in the context of task scheduling, I provide a novel reward mechanism employing a specifically created task execution cost simulator. The weight vector returned by the XCS is used to create an evaluation function for list scheduling algorithms. In an evaluation scenario consisting of an application pattern that is repeated five times, this approach decreased the makespan by $10.4\,\%$ or $26.7\,s$, the energy consumption by $4.7\,\%$ or $2061.1\,J$, and the maximum temperature of the GPU by $3.6\,\%$ or $2.7\,K$. Solely the maximum CPU core temperature was increased by $6\,\%$ or $2.3\,K$. Another important goal of the XCS approach was to guarantee a fast response time, i.e., to minimize the additional overhead created by the framework. As the generation of new rules is executed offline, i.e., in parallel to the live system, the main factor affecting the overhead is the online reward mechanism. For the five task graphs of the evaluation scenario, an average overhead of $0.115\,s$ was measured which amounts to an additional run of the HEFT algorithm. This is usually neglectable compared to the overall makespan and the potential schedule improvements.

Finally, I contributed to the research field of dynamic task scheduling in heterogeneous systems in Chapter 9 as it provides an appropriate way to influence and guide the system behavior. Particularly, I introduced two scheduling algorithms or rather extensions to scheduling algorithms. The first algorithm is specifically created for non-safety critical embedded systems that deploy task priorities to express differing importance of applications. As priorities may lead to starvation issues, I developed an aging mecha-

nism that dynamically adapts task priorities to avoid starvation and reduce overall waiting times. To evaluate the benefits of my aging mechanism, I tested it with six different dynamic scheduling algorithms in the runtime system EMB^2. In two scenarios, the average makespan was improved by $3.75\,\%$ and $2.16\,\%$ for five out of six and four out of six scheduling algorithms, respectively, while simultaneously reducing the waiting time of the lower priority tasks. This is reflected in a decrease of the average flow time (a metric for the total time a task spends in the system) of up to $25\,\%$. It has to be noted, though, that this decrease comes with an increase in flow time and individual makespan for the higher priority tasks. However, the intended hierarchy of the applications still remains intact.

The second scheduling algorithm contributes to scenarios where multiple different processes share the same processing units. This research field is called co-scheduling. My approach utilizes shared memory to share the necessary scheduling information between multiple instances of the runtime system HALadapt. When new tasks arrive in the system, my approach decides if computing a new co-schedule is reasonable. Thereby, the new tasks are combined with all tasks that have not yet started executing and a new schedule for all these tasks is computed. As a scheduling algorithm to compute the co-schedule, I deploy simulated annealing. I tested my approach in two scenarios, a CPU-only scenario with OpenMP tasks sharing 24 cores and a heterogeneous scenario with an NVIDIA Tesla K80 and the same multicore cpu. For the two CPU-only tests, my co-scheduling approach achieved average speedups of $19.52\,\%$ and $20.91\,\%$ compared to the execution without a co-scheduling mechanism. However, it has to be noted that co-scheduling several tasks on the same CPU does not generally work for all tasks as, e.g., available cache memory may be a limiting factor that can slow down parallelly executing tasks. In the heterogeneous scenario, my approach again showed its benefits and resulted in an average speedup of $17.71\,\%$.

In conclusion, the contributions of this thesis provide mechanisms that, combined, enable a system to dynamically and proactively adapt to new situations and efficiently utilize its resources concerning application makespan, energy consumption, processing unit temperatures, and system reliability. Simultaneously, the contributions reduce complexity for users and application developers by outsourcing decisions to the system itself. Thereby, usability of heterogeneous systems in different fields of application is increased.

10.2 Outlook & Future Work

Predicting OpenCL kernel execution times utilizing machine learning and static code analysis to avoid profiling runs did not result in reliable predictions. However, the approach definitely showed potential, particularly by correctly classifying kernels to the fastest processing unit and by predicting accurate execution times for some kernels. An approach for future research to improve and stabilize the prediction results is to analyze

the characteristics of kernels whose prediction values are accurate and characteristics of kernels whose prediction values are inaccurate. If there were kernel characteristics that distinguish accurate from inaccurate predictions, it would be possible to classify new kernels and only utilize the prediction mechanism if their characteristics indicate accurate prediction results.

The prediction mechanisms introduced in Chapter 7 show that it is possible to predict upcoming tasks if regular patterns exist. So far, however, they have only been evaluated in simulations. In future work, such mechanisms should be integrated into the runtime system HALadapt to combine prediction with its profiling modules. This allows to evaluate the prediction mechanisms in a running system with real applications and their impact on the XCS. Additionally, more sophisticated prediction models, e.g., created by deep neural networks may allow to predict several steps into the future or additional information.

I created a co-scheduling mechanism to utilize the high degree of parallelism offered by modern computing systems even if a single application or task may not benefit from the increased number of processing units because of limited scaling capability. Extending the co-scheduling mechanism with the approach to dynamically balance multiple optimization goals is a promising direction for future research. Furthermore, the results of the co-scheduling evaluation showed that not all task/application combinations are able to benefit from parallel execution on a multicore CPU. Therefore, studying the characteristics of applications that can be combined via co-scheduling is an interesting possibility for future work.

In the scope of this thesis, the evaluation of the dynamic and proactive adaptation of the system's optimization goals is conducted on a single system. This system is a server utilized to execute parallel and heterogeneous computations with a higher degree of parallelism. Hence, it has some basic conditions, e.g., it has sufficient cooling capacity, no energy budget limitations or is not heavily influenced by its environment as its environment is steady and optimized for the server. Therefore, further studying and analyzing the contributions of my thesis on systems with different basic conditions like embedded systems with their limited resources should lead to further insights and show additional benefits and limitations of my mechanisms. Due to the wide range of fields where my approach can be employed, more extensive evaluations with a wider range of applications from different fields are necessary to fully understand its applicability in different computing domains. In particular, embedded systems and the specific characteristics of their applications should be the focus of future work. The organic computing framework and particularly the XCS used in this thesis are defined by a set of parameters. The focus of this work is to study and analyze the ability of this framework to implement proactive system adaptation in the context of task scheduling. Hence, there is no extensive analysis of the XCS' parameters. Future work should gain insight on their influence and study the potential of adapting and optimizing these parameters.

BIBLIOGRAPHY

[1] *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems, revision 3*. (last visit 03/30/22) https://www.khronos.org/opencl/.

[2] *OpenMP - Application Programming Interface for Parallel Programming, version 5.0*. (last visit 03/30/22) https://www.openmp.org/.

[3] *MPI: A Message-Passing Interface Standard, version 4.0*. (last visit 03/30/22) https://www.mpi-forum.org/mpi-40/.

[4] *SYCL 2020, version 2.2*. (last visit 03/30/22) https://www.khronos.org/sycl/.

[5] Mario Kicherer, Fabian Nowak, et al. "Seamlessly Portable Applications: Managing the Diversity of Modern Heterogeneous Systems". In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012), 42:1–42:20. ISSN: 1544-3566.

[6] Cédric Augonnet, Samuel Thibault, et al. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience*. Euro-Par 2009 best papers 23.2 (2011), pp. 187–198.

[7] *HSA - Heterogeneous System Architecture, version 1.2*. (last visit 03/30/22) http://www.hsafoundation.com/.

[8] Scott Camazine, Nigel R. Franks, et al. *Self-Organization in Biological Systems*. Princeton, NJ, USA: Princeton University Press, 2001. ISBN: 0691012113.

[9] Christian Müller-Schloer and Sven Tomforde. "Organic Computing – Technical Systems for Survival in the Real World". In: *Autonomic Systems*. 2017.

[17] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232.

[18] G. E. Moore. “Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 36–37. ISSN: 1098-4232.

[19] R. H. Dennard, F. H. Gaensslen, et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 1558-173X.

[20] Wm. A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[22] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956.

[23] Sparsh Mittal and Jeffrey S. Vetter. “A Survey of CPU-GPU Heterogeneous Computing Techniques”. In: *ACM Comput. Surv.* 47.4 (July 2015). ISSN: 0360-0300.

[24] K. Pocek, R. Tessier, and A. DeHon. “Birth and adolescence of reconfigurable computing: a survey of the first 20 years of field-programmable custom computing machines”. In: *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. Apr. 2013, pp. 1–17.

[25] *CUDA Toolkit Documentation, version 10.2.89*. (last visit 03/30/22) `https://docs.nvidia.com/cuda/index.html`.

[26] *1800–2017 – IEEE Standard for SystemVerilog – Unified Hardware Desing, Specification, and Verification Language*. (last visit 03/30/22) `https://standards.ieee.org/standard/1800-2017.html`.

[27] “IEEE Standard VHDL Language Reference Manual”. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan. 2009), pp. 1–640. ISSN: null.

[28] *The OpenACC Application Programming Interface, version 3.0*. (last visit 03/30/22) `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf`.

[29] Karim Djemame, Django Armstrong, et al. “TANGO: Transparent heterogeneous hardware Architecture deployment for eNergy Gain in Operation”. In: *CoRR* abs/1603.01407 (2016). arXiv: `1603.01407`.

[30] *Xilinx Zynq Ultrascale+ MPSoC Data Sheet: Overview, version 1.8*. (last visit 03/30/22) https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[31] *Get Under the Hood of Parker, Our Newest SOC for Autonomous Vehicles*. (last visit 03/30/22) https://blogs.nvidia.com/blog/2016/08/22/parker-for-self-driving-cars/.

[32] George Vlahakis and Dimitris Apostolou. "Proactivity in Service Based Applications". In: *Proceedings of the 2012 16th Panhellenic Conference on Informatics*. PCI '12. USA: IEEE Computer Society, 2012, pp. 62–67. ISBN: 9780769548258.

[33] V. Klös, T. Göthel, and S. Glesner. "Be Prepared: Learning Environment Profiles for Proactive Rule-Based Production Planning". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2018, pp. 89–96.

[34] A. Bousdekis, N. Papageorgiou, et al. "A probabilistic model for context-aware proactive decision making". In: *2016 7th International Conference on Information, Intelligence, Systems Applications (IISA)*. July 2016, pp. 1–6.

[35] Gregory D. Abowd, Anind K. Dey, et al. "Towards a Better Understanding of Context and Context-Awareness". In: *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*. HUC '99. Karlsruhe, Germany: Springer-Verlag, 1999, pp. 304–307. ISBN: 3540665501.

[36] Kai Tian, Yunlian Jiang, et al. "An Input-Centric Paradigm for Program Dynamic Optimizations". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 125–139. ISBN: 9781450302036.

[37] J. Grosinger, F. Pecora, and A. Saffiotti. "Proactivity through equilibrium maintenance with fuzzy desirability". In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Oct. 2017, pp. 2117–2122.

[38] Geoffroy Vallee, Kulathep Charoenpornwattana, et al. "A Framework for Proactive Fault Tolerance". In: *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*. ARES '08. USA: IEEE Computer Society, 2008, pp. 659–664. ISBN: 9780769531021.

[39] D. Kramer and W. Karl. "Realizing a Proactive, Self-Optimizing System Behavior within Adaptive, Heterogeneous Many-Core Architectures". In: *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*. Sept. 2012, pp. 39–48.

[40] Yagil Engel and Opher Etzion. “Towards Proactive Event-Driven Computing”. In: *Proceedings of the 5th ACM International Conference on Distributed Event-Based System*. DEBS ’11. New York, New York, USA: Association for Computing Machinery, 2011, pp. 125–136. ISBN: 9781450304238.

[41] Sebastian VanSyckel. “System Support for Proactive Adaptation”. PhD thesis. University of Mannheim, 2015. URL: https://ub-madoc.bib.uni-mannheim.de/39016.

[42] Jochen Fromm. *The emergence of complexity*. First. Kassel University Press GmbH, 2004. ISBN: 978-3-89958-069-3.

[43] Sven A. Brueckner and Hans Czap. “Organization, Self-Organization, Autonomy and Emergence: Status and Challenges”. In: *ITSSA* 2 (2006), pp. 1–10.

[44] Wilfried Elmenreich and Hermann de Meer. “Self-Organizing Networked Systems for Technical Applications: A Discussion on Open Issues”. In: *Self-Organizing Systems*. Ed. by Karin Anna Hummel and James P. G. Sterbenz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–9. ISBN: 978-3-540-92157-8.

[45] Tom De Wolf and Tom Holvoet. “Emergence Versus Self-Organisation: Different Concepts but Promising When Combined”. In: *Engineering Self-Organising Systems*. Ed. by Sven A. Brueckner, Giovanna Di Marzo Serugendo, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–15. ISBN: 978-3-540-31901-6.

[46] Francis Heylighen. “Complexity and Self-organization”. In: *Encyclopedia of Library and Information Sciences* (Jan. 2008).

[47] H. Schmeck. “Organic computing - a new vision for distributed embedded systems”. In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’05)*. May 2005, pp. 201–203.

[48] C. Müller-Schloer. “Organic Computing: On the Feasibility of Controlled Emergence”. In: *Proceedings of the 2Nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’04. Stockholm, Sweden: ACM, 2004, pp. 2–5. ISBN: 1-58113- 937-3.

[49] U. Richter, M. Mnif, et al. “Towards a Generic Observer/Controller Architecture for Organic Computing”. In: *Informatik 2006, Informatik für Menschen. Ed.: C. Hochberger*. Vol. 93. GI-Edition, lecture notes in informatics. Gesellschaft für Informatik, Bonn, 2006, pp. 112–119. ISBN: 978-3-88579-187-4.

[50] Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. *Organic Computing - A Paradigm Shift for Complex Systems*. 1st. Berlin, Heidelberg: Springer-Verlag, 2011. ISBN: 3034801297, 9783034801294.

[51] Sven Tomforde, Bernhard Sick, and Christian Müller-Schloer. “Organic Computing in the Spotlight”. In: *CoRR* abs/1701.08125 (2017). arXiv: 1701.08125.

[52] R. Allrutz, C. Cap, et al. “Organic Computing – Computer- und Systemarchitektur im Jahr 2010”. In: *VDE/ITG/GI-Positionspapier* (2003).

[53] Thorsten Schöler and Christian Müller-Schloer. “An Observer/Controller Architecture for Adaptive Reconfigurable Stacks”. In: vol. 3432. Mar. 2005, pp. 139–153.

[54] J.J. Di Stefano, A.R. Stubberud, and I.J. Williams. *Feedback and Control Systems*. Schaum’s outline series. McGraw-Hill Book Company, 1976.

[55] J. O. Kephart and D. M. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (Jan. 2003), pp. 41–50.

[56] S. R. White, J. E. Hanson, et al. “An architectural approach to autonomic computing”. In: *International Conference on Autonomic Computing, 2004. Proceedings.* May 2004, pp. 2–9.

[57] H. Choset, K. M. Lynch, et al. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MITP, 2005. ISBN: 9780262255912.

[58] Sven Burmester, Holger Giese, et al. “Tool support for the design of self-optimizing mechatronic multi-agent systems”. In: *International Journal on Software Tools for Technology Transfer* 10.3 (June 2008), pp. 207–222. ISSN: 1433-2787.

[59] Michael Wooldridge. *An Introduction to MultiAgent Systems*. 2nd. Wiley Publishing, 2009. ISBN: 0470519460.

[60] Mario Kicherer. “Reducing the Complexity of Heterogeneous Computing: A Unified Approach for Application Development and Runtime Optimization”. PhD thesis. 2014.

[61] Mario Kicherer and Wolfgang Karl. “Automatic task mapping and heterogeneity-aware fault tolerance: The benefits for runtime optimization and application development”. In: *Journal of Systems Architecture* 61.10 (2015). Special section on Architecture of Computing Systems edited by Editors: Wolfgang Karl, Erik Maehle, Kay Römer, Eduardo Tovar, Martin Danek Special section on Testing, Prototyping, and Debugging of Multi-Core Architectures edited by Editors: Frank Hannig & Andreas Herkersdorf Special section on Embedded Vision Architectures and Applications edited by Editors: Christophe Bobda, Walter Stechele, Ali Ahmadinia and Miaoqing Huang, pp. 628–638. ISSN: 1383-7621.

[62] Mario Kicherer, Rainer Buchty, and Wolfgang Karl. “Cost-aware Function Migration in Heterogeneous Systems”. In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. HiPEAC 11. Heraklion, Greece: ACM, 2011, pp. 137–145. ISBN: 978-1-4503-0241-8.

[63] Tobias Schuele. “Embedded Multicore Building Blocks: Parallel Programming Made Easy”. In: *Embedded World* (2015).

[64] Urs Gleim and Markus Levy. *MTAPI: Parallel Programming for Embedded Multicore Systems*. 2013. URL: http://multicore-association.org/pdf/MTAPI%5C_Overview%5C_2013.pdf.

[65] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *Journal of the ACM* 46.5 (Sept. 1999), pp. 720–748.

[66] Sebastian Mattheis, Tobias Schuele, et al. “Work Stealing Strategies for Parallel Stream Processing in Soft Real-time Systems”. In: *Proceedings of the 25th International Conference on Architecture of Computing Systems*. ARCS 12. Munich, Germany: Springer-Verlag, 2012, pp. 172–183. ISBN: 978-3-642-28292-8.

[67] Alejandro Duran, Eduard Ayguadé, et al. “Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures”. In: *Parallel Processing Letters* 21 (2011), pp. 173–193.

[68] Eduard Ayguadé, Rosa M. Badia, et al. “Extending OpenMP to Survive the Heterogeneous Multi-Core Era”. In: *International Journal of Parallel Programming* 38.5 (Oct. 2010), pp. 440–459. ISSN: 1573-7640.

[69] *OmpSs-2 Specification*. (last visit 03/30/22) https://pm.bsc.es/ftp/ompss-2/doc/spec/.

[70] *Nanos++ Runtime*. (last visit 03/30/22) https://github.com/bsc-pm/nanox.

[71] *Nanos6 Runtime*. (last visit 03/30/22) https://github.com/bsc-pm/nanos6.

[72] J. M. Perez, R. M. Badia, and J. Labarta. “A dependency-aware task-based programming environment for multi-core architectures”. In: *2008 IEEE International Conference on Cluster Computing*. Sept. 2008, pp. 142–151.

[73] Judit Planas, Rosa Maria Badia, et al. “Selection of Task Implementations in the Nanos++ Runtime”. In: 2013.

[74] TANGO Whitepaper. “TANGO Toolbox-Final version scientific report”. In: 2019.

[75] *TANGO Architecture*. (last visit 03/30/22) http://www.tango-project.eu/sites/default/files/tango/public/field/image/TANGO_architecture.jpg.

[76] Rosa M. Badia, Javier Conejero, et al. “COMP Superscalar, an interoperable programming framework”. In: *SoftwareX* 3-4 (2015), pp. 32–36. ISSN: 2352-7110.

[77] R. Kavanagh and K. Djemame. “Energy-aware Self-Adaptive Middleware for Heterogeneous Parallel Architectures”. In: *2018 Fifth International Symposium on Innovation in Information and Communication Technology (ISIICT)*. Oct. 2018, pp. 1–8.

[78] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. “Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures”. In: *Euro-Par 2009 – Parallel Processing Workshops*. Ed. by Hai-Xiang Lin, Michael Alexander, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 56–65. ISBN: 978-3-642-14122-5.

[79] Cédric Augonnet and Raymond Namyst. “A Unified Runtime System for Heterogeneous Multi-core Architectures”. In: *Euro-Par 2008 Workshops - Parallel Processing*. Ed. by Eduardo César, Michael Alexander, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–183. ISBN: 978-3-642-00955-6.

[80] Thomas Heller, Patrick Diehl, et al. “HPX – An open source C++ Standard Library for Parallelism and Concurrency”. In: *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017 (OpenSuCo’17)*. 2017, p. 5.

[81] P. Diehl, M. Seshadri, et al. “Integration of CUDA Processing within the C++ Library for Parallelism and Concurrency (HPX)”. In: *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. Nov. 2018, pp. 19–28.

[82] M. Bauer, S. Treichler, et al. “Legion: Expressing locality and independence with logical regions”. In: *SC 12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2012, pp. 1–11.

[83] K. Bergman, S. Borkar, et al. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead*. 2008.

[84] Sparsh Mittal. “Power Management Techniques for Data Centers: A Survey”. In: (Apr. 2014).

[85] Edward Curry, Bill Guyon, et al. “Developing an Sustainable IT Capability: Lessons From Intel’s Journey”. In: *MIS Quarterly Executive* 11 (June 2012), pp. 61–74.

[86] Tobias Schuele. “Embedded Multicore Building Blocks: Parallel Programming Made Easy”. In: *Embedded World* (2015).

[87] Renhan Lou. “Symptom-based Fault Detection on GPUs”. Bachelor’s Thesis. Karlsruhe Institute of Technology, May 2020.

[88] Markus Helwig. “Leistungsvorhersage OpenCL-fähiger Beschleuniger mittels statischer Code-Analyse und Machine Learning”. MA thesis. Karlsruhe Institute of Technology, Feb. 2017.

[89] D. Kramer and W. Karl. “Realizing a Proactive, Self-Optimizing System Behavior within Adaptive, Heterogeneous Many-Core Architectures”. In: *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*. Sept. 2012, pp. 39–48.

[90] *Intel 64 and IA-32 Architectures Software Developer's Manual, volume 1, 2ABCD, 3ABCD, 4*. (last visit 03/30/22) `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`.

[91] *AMD uProf User Guide, version 3.2*. (last visit 03/30/22) `https://developer.amd.com/wordpress/media/files/AMDuprof_Resources/User_Guide_AMD_uProf_v3.2_GA.pdf`.

[92] *NVIDIA Management Library*. (last visit 03/30/22) `https://developer.nvidia.com/nvidia-management-library-nvml`.

[93] *Linux manual page perf_event_open*. (last visit 03/30/22) `http://man7.org/linux/man-pages/man2/perf_event_open.2.html`.

[94] *lm-sensors*. (last visit 03/30/22) `https://github.com/lm-sensors/lm-sensors`.

[95] Dan Terpstra, Heike Jagode, et al. “Collecting Performance Data with PAPI-C”. In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller, Michael M. Resch, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173. ISBN: 978-3-642-11261-4.

[96] Jan Treibig, Georg Hager, and Gerhard Wellein. “LIKWID: Lightweight Performance Tools”. In: *Competence in High Performance Computing 2010*. Ed. by Christian Bischof, Heinz-Gerd Hegering, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 165–175. ISBN: 978-3-642-24025-6.

[97] *likwid-perfctr: Measuring applications' interaction with the hardware using the hardware performance counters*. (last visit 03/30/22) `https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr`.

[98] *CUPTI documentation*. last visit (03/30/22) `https://docs.nvidia.com/cupti/Cupti/index.html`. NVIDIA Corporation, 2007–2019.

[99] John Wernsing and Greg Stitt. “Elastic Computing: A Framework for Transparent, Portable, and Adaptive Multi-core Heterogeneous Computing”. In: vol. 45. Apr. 2010, pp. 115–124.

[100] Artur Podobas, Mats Brorsson, and Vladimir Vlassov. “Exploring Heterogeneous Scheduling Using the Task-Centric Programming Model”. In: *Euro-Par 2012: Parallel Processing Workshops*. Ed. by Ioannis Caragiannis, Michael Alexander, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 133–144. ISBN: 978-3-642-36949-0.

[101] Víctor J. Jiménez, Lluís Vilanova, et al. “Predictive Runtime Code Scheduling for Heterogeneous Architectures”. In: *High Performance Embedded Architectures and Compilers*. Ed. by André Seznec, Joel Emer, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 19–33. ISBN: 978-3-540-92990-1.

[102] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. “MDR: Performance Model Driven Runtime for Heterogeneous Parallel Platforms”. In: *Proceedings of the International Conference on Supercomputing*. ICS ’11. Tucson, Arizona, USA: ACM, 2011, pp. 225–234. ISBN: 978-1-4503-0102-2.

[103] *What exactly is a P-State?* (last visit 03/30/22) `https://software.intel.com/en-us/blogs/2008/05/29/what-exactly-is-a-p-state-pt-1`.

[104] *intel_pstate*. (last visit 03/30/22) `https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html`.

[105] Felix Salfner, Maren Lenk, and Miroslaw Malek. “A Survey of Online Failure Prediction Methods”. In: *ACM Comput. Surv.* 42.3 (Mar. 2010), 10:1–10:42. ISSN: 0360-0300.

[106] Wolfram Schiffmann and Robert Schmitz. *Technische Informatik 2*. Springer Berlin Heidelberg, Jan. 2002.

[107] A. W. Williams, S. M. Pertet, and P. Narasimhan. “Tiresias: Black-Box Failure Prediction in Distributed Systems”. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. Mar. 2007, pp. 1–8.

[108] Joy Arulraj, Po-Chun Chang, et al. “Production-run Software Failure Diagnosis via Hardware Performance Counters”. In: *SIGARCH Comput. Archit. News* 41.1 (Mar. 2013), pp. 101–112. ISSN: 0163-5964.

[109] Cemal Yilmaz and Adam Porter. “Combining Hardware and Software Instrumentation to Classify Program Executions”. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE ’10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 67–76. ISBN: 978-1-60558-791-2.

[110] Martin Dimitrov and Huiyang Zhou. “Unified Architectural Support for Soft-Error Protection or Software Bug Detection”. In: Oct. 2007, pp. 73–82. ISBN: 978-0-7695-2944-8.

[111] S. Narayanasamy, A. K. Coskun, and B. Calder. “Transient Fault Prediction Based on Anomalies in Processor Events”. In: *2007 Design, Automation Test in Europe Conference Exhibition*. Apr. 2007, pp. 1–6.

[112] N. J. Wang and S. J. Patel. “ReStore: symptom based soft error detection in microprocessors”. In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. June 2005, pp. 30–39.

[113] S. K. S. Hari, M. Li, et al. “mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems”. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2009, pp. 122–132.

[114] C. Ding, C. Karlsson, et al. “Matrix Multiplication on GPUs with On-Line Fault Tolerance”. In: *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. May 2011, pp. 311–317.

[115] N. Maruyama, A. Nukada, and S. Matsuoka. “A high-performance fault-tolerant software framework for memory on commodity GPUs”. In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. Apr. 2010, pp. 1–12.

[116] S. Di Carlo, G. Gambardella, et al. “A software-based self test of CUDA Fermi GPUs”. In: *2013 18th IEEE European Test Symposium (ETS)*. May 2013, pp. 1–6.

[117] Jingweijia Tan and Xin Fu. “RISE: Improving the streaming processors reliability against soft errors in GPGPUs”. In: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 2012, pp. 191–200.

[118] Alessio Netti, Zeynep Kiziltan, et al. “FINJ: A Fault Injection Tool for HPC Systems”. In: *CoRR* abs/1807.10056 (2018). arXiv: 1807.10056.

[119] S. K. S. Hari, T. Tsai, et al. “SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation”. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2017, pp. 249–258.

[120] *SASSIFI*. (last visit 03/30/22) https://github.com/NVlabs/sassifi.

[121] M. Stephenson, S. K. S. Hari, et al. “Flexible software profiling of GPU architectures”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 185–197.

[122] Ozan Tuncer, Emre Ates, et al. “Diagnosing Performance Variations in HPC Applications Using Machine Learning”. In: *High Performance Computing*. Ed. by Julian M. Kunkel, Rio Yokota, et al. Cham: Springer International Publishing, 2017, pp. 355–373. ISBN: 978-3-319-58667-0.

[123] Shuai Che, Michael Boyer, et al. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. IISWC 09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. ISBN: 978-1-4244-5156-2.

[124] B. L. WELCH. “THE GENERALIZATION OF ’STUDENT’S’ PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED”. In: *Biometrika* 34.1-2 (Jan. 1947), pp. 28–35. ISSN: 0006-3444. eprint: http://oup.prod.sis.lan/biomet/article-pdf/34/1-2/28/553093/34-1-2-28.pdf.

[125] Microsoft. *Microsoft Excel – TTEST-Function*. 2020. URL: https://support.microsoft.com/de-de/office/ttest-funktion-1696ffc1-4811-40fd-9d13-a0eaad83c7ae.

[126] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, Mar. 2004.

[127] Brian A. Nejmeh. “NPATH: A Measure of Execution Path Complexity and Its Applications”. In: *Commun. ACM* 31.2 (Feb. 1988), pp. 188–200. ISSN: 0001-0782.

[128] T. J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 0098-5589.

[129] OCLint. *OCLint*. (last visit 03/30/22) http://oclint.org/. 2012.

[130] Jason R. C. Patterson. “Accurate Static Branch Prediction by Value Range Propagation”. In: *SIGPLAN Not.* 30.6 (June 1995), pp. 67–78. ISSN: 0362-1340.

[131] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. 2nd. Chapman & Hall/CRC, 2014. ISBN: 1466583282, 9781466583283.

[132] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN: 026201825X, 9780262018258.

[133] T. A. Runkler. *Data Analytics - Models and Algorithms for Intelligent Data Analysis, Second Edition*. Springer Wiesbaden, 2016. ISBN: 978-3-658-14075-5.

[134] Tin Kam Ho. “Random Decision Forests”. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*. ICDAR ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 278–. ISBN: 0-8186-7128-9.

[135] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. Springer, 2009.

[136] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[137] Paul Munro. "Backpropagation". In: *Encyclopedia of Machine Learning*. Springer, 2010. ISBN: 978-0-387-30164-8.

[138] F. Pedregosa, G. Varoquaux, and A. Gramfort et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* (2011).

[139] AMD. *AMD APP SDK*. (last visit 09/30/16) `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`. 2016.

[140] Intel. *Intel OpenCL Samples*. (last visit 03/30/22) `https://software.intel.com/en-us/intel-opencl-support/code-samples`. 2016.

[141] Y. Sun, X. Gong, et al. "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing". In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2016.

[142] J. A. Stratton, C. Rodrigues, et al. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Tech. rep. IMPACT-12-01. University of Illinois at Urbana-Champaign, Mar. 2012.

[143] S. Grauer-Gray, L. Xu, et al. "Auto-tuning a high-level language targeted to GPU codes". In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pp. 1–10.

[144] Scott Grauer-Gray and Louis-Noel Pouchet. *PolyBench/GPU - Implementation of PolyBench codes for GPU processing*. (last visit 03/30/22) `http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/`. Mar. 2012.

[145] A. Danalis, G. Marin, et al. "The Scalable Heterogeneous Computing (SHOC) benchmark suite." In: *GPGPU*. Vol. 425. ACM International Conference Proceeding Series. Mar. 18, 2010.

[146] C. Beffa and R. J Connell. "Two-dimensional flood plain flow. I: Model description". In: *Journal of Hydrologic Engineering* (2001).

[147] Tomas Borovicka, Marcel Jirina Jr, et al. *Selecting representative data sets*. INTECH Open Access Publisher, 2012.

[148] Yi-Wei Chen and Chih-Jen Lin. "Combining SVMs with Various Feature Selection Strategies". In: *Feature Extraction: Foundations and Applications*. Ed. by Isabelle Guyon, Masoud Nikravesh, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 315–324. ISBN: 978-3-540-35488-8.

[149] I. Baldini, S. J. Fink, and E. R. Altman. "Predicting GPU Performance from CPU Runs Using Machine Learning." In: *SBAC-PAD*. IEEE Computer Society, 2014, pp. 254–261. ISBN: 978-1-4799-6904-3.

[150] Y. Wen, Z. Wang, and M. F. P. O'Boyle. "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms". In: *2014 21st International Conference on High Performance Computing (HiPC)*. Dec. 2014, pp. 1–10.

[151] A. Hayashi, K. Ishizaki, et al. "Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ '15. Melbourne, FL, USA: ACM, 2015, pp. 27–36. ISBN: 978-1-4503-3712-0.

[152] G. Wu, J. L. Greathouse, et al. "GPGPU performance and power estimation using machine learning". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2015, pp. 564–576.

[153] M. Amarís, R. Y. de Camargo, et al. "A comparison of GPU execution time prediction using machine learning and analytical modeling". In: *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. Oct. 2016, pp. 326–333.

[154] K. Hoste, A. Phansalkar, et al. "Performance prediction based on inherent program similarity". In: *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2006, pp. 114–122.

[155] D. Grewe and M. F. P. O'Boyle. "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL". In: *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*. CC'11/ETAPS'11. Saarbrücken, Germany: Springer-Verlag, 2011, pp. 286–305. ISBN: 978-3-642-19860-1.

[156] K. Kofler, I. Grasso, et al. "An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning". In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. Eugene, Oregon, USA, 2013. ISBN: 978-1-4503-2130-3.

[157] Spyros Makridakis. "Accuracy measures: theoretical and practical concerns". In: *International Journal of Forecasting* 9.4 (1993), pp. 527–529. ISSN: 0169-2070.

[158] Jan Petzold, Faruk Bagci, et al. "The state predictor method for context prediction". In: *Proceedings: Fifth International Conference on Ubiquitous Computing 2003 (UbiComp 2003) October 12-15, 2003, Seattle, Washington, USA*. Ed. by Joe McCarthy, Anind K. Dey, and Albrecht Schmidt. 2003, pp. 191–192.

[159] A. E. Nicholson and J. M. Brady. "Dynamic belief networks for discrete monitoring". In: *IEEE Transactions on Systems, Man, and Cybernetics* 24.11 (Nov. 1994), pp. 1593–1610. ISSN: 2168-2909.

[160] Uri Nodelman, Christian R. Shelton, and Daphne Koller. "Continuous Time Bayesian Networks". In: UAI'02. Alberta, Canada: Morgan Kaufmann Publishers Inc., 2002, pp. 378–387. ISBN: 1558608974.

[161] Christos Anagnostopoulos, Panagiotis Mpougiouris, and Stathes Hadjiefthymiades. “Prediction Intelligence in Context-Aware Applications”. In: *Proceedings of the 6th International Conference on Mobile Data Management*. MDM ’05. Ayia Napa, Cyprus: Association for Computing Machinery, 2005, pp. 137–141. ISBN: 1595930418.

[162] Fenghua Gao, J. S. Thorp, et al. “Dynamic state prediction based on Auto-Regressive (AR) Model using PMU data”. In: *2012 IEEE Power and Energy Conference at Illinois*. Feb. 2012, pp. 1–5.

[163] H. Livani, S. Jafarzadeh, et al. “A Unified Approach for Power System Predictive Operations Using Viterbi Algorithm”. In: *IEEE Transactions on Sustainable Energy* 5.3 (July 2014), pp. 757–766. ISSN: 1949-3037.

[164] L. Hernandez, C. Baladron, et al. “A multi-agent system architecture for smart grid management and forecasting of energy demand in virtual power plants”. In: *IEEE Communications Magazine* 51.1 (Jan. 2013), pp. 106–113. ISSN: 1558-1896.

[165] Michael Hind, Vadakkedathu Rajan, and Peter Sweeney. “Phase Shift Detection: A Problem Classification”. In: (Dec. 2003).

[166] Thomas Becker. “Phasendefinition und Phasenvorhersage in adaptiven Many-Core Architekturen”. Bachelor’s Thesis. Karlsruhe Institute of Technology, Sept. 2011.

[167] David Kramer. “Self-awareness in heterogeneous, adaptive many-core architectures enabling proactive, self-optimizing systems”. Zsfassungen in dt. und engl. SpracheZugl.: Karlsruhe, Karlsruher Inst. fúr Technologie, Diss., 2012IMD-Felder maschinell generiert (GBV). PhD thesis. Aachen, 2012. ISBN: 9783844013450. URL: https://www.gbv.de/dms/tib-ub-hannover/725615893.pdf.

[168] Frederik Vandeputte, Lieven Eeckhout, and Koen De Bosschere. “A Detailed Study on Phase Predictors”. In: Euro-Par’05. Lisbon, Portugal: Springer-Verlag, 2005, pp. 571–581. ISBN: 3540287000.

[169] Timothy Sherwood, Suleyman Sair, and Brad Calder. “Phase Tracking and Prediction”. In: 31.2 (May 2003), pp. 336–349. ISSN: 0163-5964.

[170] Martin Hock, Karthik Jayaraman, et al. *Phase Capture and Prediction with Applications*. 2005.

[171] Xipeng Shen, Yutao Zhong, and Chen Ding. “Locality Phase Prediction”. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XI. Boston, MA, USA: Association for Computing Machinery, 2004, pp. 165–176. ISBN: 1581138040.

[172] Xipeng Shen, Yutao Zhong, and Chen Ding. “Predicting locality phases for dynamic memory optimization”. In: *J. Parallel Distrib. Comput* (2007), pp. 783–796.

[173] Zhenman Fang, Jiaxin Li, et al. “Improving Dynamic Prediction Accuracy through Multi-Level Phase Analysis”. In: LCTES ’12. Beijing, China: Association for Computing Machinery, 2012, pp. 89–98. ISBN: 9781450312127.

[174] H. Quan, T. Zhang, and J. Guo. “Task Scheduling Prediction Algorithms for Dynamic Hardware/Software Partitioning”. In: *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*. Dec. 2012, pp. 80–85.

[175] Andrey Vladimirovich Gritsenko, Nikita Georgievich Demurchev, et al. “Decomposition Analysis and Machine Learning in a Workflow-Forecast Approach to the Task Scheduling Problem for High-Loaded Distributed Systems”. In: *Mathematical Models and Methods in Applied Sciences* 9 (2015), p. 38.

[176] George Edward Pelham Box and Gwilym Jenkins. *Time Series Analysis, Forecasting and Control*. USA: Holden-Day, Inc., 1990. ISBN: 0816211043.

[177] Hema R. Madala and Alekseæi Grigoşevich Ivakhnenko. *Inductive Learning Algorithms for Complex Systems Modeling*. USA: CRC Press, Inc., 1994. ISBN: 0849344387.

[178] N. Golyandina and D. Stepanov. “SSA-based approaches to analysis and forecast of multidimensional time series”. In: *Proceedings of the 5th St. Petersburg Workshop on Simulation*. 2005.

[179] “Reprint of: Mahalanobis, P.C. (1936) "On the Generalised Distance in Statistics."” In: *Sankhya A* 80.1 (Dec. 2018), pp. 1–7. ISSN: 0976-8378.

[180] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. AMS, 2003.

[181] Xinye Yang. “Markov Chain and Its Applications”. PhD thesis. Mar. 2020.

[182] D. Joseph and D. Grunwald. “Prefetching using Markov predictors”. In: *IEEE Transactions on Computers* 48.2 (Feb. 1999), pp. 121–133. ISSN: 1557-9956.

[183] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. “Analysis of Branch Prediction via Data Compression”. In: ASPLOS VII. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1996, pp. 128–137. ISBN: 0897917677.

[184] Robert I. Davis and Alan Burns. “A Survey of Hard Real-Time Scheduling for Multiprocessor Systems”. In: *ACM Comput. Surv.* 43.4 (Oct. 2011). ISSN: 0360-0300.

[185] Leonard Kleinrock. *Theory, Volume 1, Queueing Systems*. USA: Wiley-Interscience, 1975. ISBN: 0471491101.

[186] *Mersenne Twister 64bit version*. (last visit 03/30/22) `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html`.

[187] Makoto Matsumoto and Takuji Nishimura. “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301.

[188] Shouvik Chakraborty and Kalyani Mali. “Application of Multiobjective Optimization Techniques in Biomedical Image Segmentation—A Study”. In: *Multi-Objective Optimization: Evolutionary to Hybrid Framework*. Ed. by Jyotsna K. Mandal, Somnath Mukhopadhyay, and Paramartha Dutta. Singapore: Springer Singapore, 2018, pp. 181–194. ISBN: 978-981-13-1471-1.

[189] Seyedali Mirjalili and Jin Song Dong. *Multi-Objective Optimization using Artificial Intelligence Techniques*. 1st. Springer International Publishing, 2020. ISBN: 3030248345.

[190] Abdullah Konak, David W. Coit, and Alice E. Smith. “Multi-objective optimization using genetic algorithms: A tutorial”. In: *Reliability Engineering & System Safety* 91.9 (2006). Special Issue - Genetic Algorithms and Reliability, pp. 992–1007. ISSN: 0951-8320.

[191] Anne Koziolek. “Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes”. PhD thesis. Karlsruhe, Germany: Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie, July 2011. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000024955.

[192] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[193] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. “Reinforcement Learning: A Survey”. In: 4.1 (May 1996), pp. 237–285. ISSN: 1076-9757.

[194] G. Mongillo, H. Shteingart, and Y. Loewenstein. “The Misbehavior of Reinforcement Learning”. In: *Proceedings of the IEEE* 102.4 (2014), pp. 528–541.

[195] John H. Holland. “Adaptation”. In: *Progress in Theoretical Biology*. New York:Plenum, 1976, pp. 263–293. ISBN: 978-0-12-543104-0.

[196] John Holland, Lashon Booker, et al. “What Is a Learning Classifier System?” In: vol. 1813. Jan. 1999, pp. 3–32. ISBN: 978-3-540-67729-1.

[197] John H. Holland. “Adaptive algorithms for discovering and using general patterns in grow ing knowledge-bases”. In: *International Journal of Policy Analysis and Information Systems* 4.3 (1980), pp. 245–268.

[198] Ryan J. Urbanowicz and Will N. Browne. *Introduction to Learning Classifier Systems*. 1st. Springer Publishing Company, Incorporated, 2017. ISBN: 3662550067.

[199] Ryan Urbanowicz and Jason Moore. “Learning Classifier Systems: A Complete Introduction, Review, and Roadmap”. In: *Journal of Artificial Evolution and Applications* 2009 (Sept. 2009).

[200] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Republished by the MIT Press, 1992. Ann Arbor: University of Michigan Press, 1975. ISBN: 0472084607.

[201] Stewart W. Wilson. “Classifier Fitness Based on Accuracy”. In: *Evol. Comput.* 3.2 (June 1995), pp. 149–175. ISSN: 1063-6560.

[202] Christopher John Cornish Hellaby Watkins. “Learning from Delayed Rewards”. PhD thesis. Cambridge, UK: King’s College, May 1989. URL: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

[203] Stewart W. Wilson. “Get Real! XCS with Continuous-Valued Inputs”. In: *Learning Classifier Systems, From Foundations to Applications*. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 209–222. ISBN: 3540677291.

[204] S. W. Wilson. “Generalization in the XCS Classifier System”. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Ed. by J. R. Koza, W. Banzhaf, et al. San Francisco, CA: Morgan Kaufmann, 1998.

[205] Stewart W. Wilson. “Classifiers That Approximate Functions”. In: *Natural Computing: An International Journal* 1.2-3 (June 2002), pp. 211–234. ISSN: 1567-7818.

[206] Y. C. Lee and A. Y. Zomaya. “Energy Conscious Scheduling for Distributed Computing Systems under Different Operating Conditions”. In: *IEEE Transactions on Parallel and Distributed Systems* 22.8 (2011), pp. 1374–1381.

[207] Q. Chen, L. Zheng, et al. “EEWA: Energy-Efficient Workload-Aware Task Scheduling in Multi-core Architectures”. In: *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 2014, pp. 642–651.

[208] Kai Ma, Xue Li, et al. “GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures”. In: Sept. 2012, pp. 48–57. ISBN: 978-1-4673-2508-0.

[209] W. Sun and T. Sugawara. “Heuristics and Evaluations of Energy-Aware Task Mapping on Heterogeneous Multiprocessors”. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 599–607.

[210] Jin Cui and Douglas L. Maskell. “Dynamic Thermal-Aware Scheduling on Chip Multiprocessor for Soft Real-Time System”. In: GLSVLSI ’09. Boston Area, MA, USA: Association for Computing Machinery, 2009, pp. 393–396. ISBN: 9781605585222.

[211] H. F. Sheikh, I. Ahmad, and D. Fan. “An Evolutionary Technique for Performance-Energy-Temperature Optimized Scheduling of Parallel Tasks on Multi-Core Processors”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.3 (2016), pp. 668–681.

[212] Holger Prothmann, Sven Tomforde, et al. “Organic Traffic Control”. In: *Organic Computing — A Paradigm Shift for Complex Systems*. Ed. by Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. Basel: Springer Basel, 2011, pp. 431–446.

[213] Björn Hurling, Sven Tomforde, and Jörg Hähner. “Organic Network Control”. In: *Organic Computing — A Paradigm Shift for Complex Systems*. Ed. by Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. Basel: Springer Basel, 2011, pp. 611–613. ISBN: 978-3-0348-0130-0.

[214] Sven Tomforde, Björn Hurling, and Jörg Hähner. “Distributed Network Protocol Parameter Adaptation in Mobile Ad-Hoc Networks”. In: *Informatics in Control, Automation and Robotics*. Ed. by Juan Andrade Cetto, Jean-Louis Ferrier, and Joaquim Filipe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 91–104. ISBN: 978-3-642-19539-6.

[215] Sven Tomforde, Ioannis Zgeras, et al. “Adaptive Control of Sensor Networks”. In: *Autonomic and Trusted Computing*. Ed. by Bing Xie, Juergen Branke, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 77–91. ISBN: 978-3-642-16576-4.

[216] Pier Luca Lanzi, Daniele Loiacono, and Pier Luca. “XCSLib: The XCS classifier system library”. In: (Apr. 2009).

[217] Sushu Zhang and Karam S. Chatha. “Approximation algorithm for the temperature-aware scheduling problem”. In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. 2007, pp. 281–288.

[218] Kevin Skadron, Mircea R. Stan, et al. “Temperature-Aware Microarchitecture: Modeling and Implementation”. In: *ACM Trans. Archit. Code Optim.* 1.1 (Mar. 2004), pp. 94–125. ISSN: 1544-3566.

[219] K. Skadron, T. Abdelzaher, and M.R. Stan. “Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management”. In: *Proceedings Eighth International Symposium on High Performance Computer Architecture*. 2002, pp. 17–28.

[220] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. “Task Scheduling Algorithms for Heterogeneous Processors”. In: *Proceedings of the Eighth Heterogeneous Computing Workshop*. HCW ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 3–. ISBN: 0-7695-0107-9.

[221] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing”. In: *IEEE Trans. Parallel Distrib. Syst.* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219.

[222] L. A. Shepp and Y. Vardi. “Maximum Likelihood Reconstruction for Emission Tomography”. In: *IEEE Transactions on Medical Imaging* 1.2 (1982), pp. 113–122.

[223] Tilman Küstner, Josef Weidendorfer, et al. “Parallel MLEM on Multicore Architectures”. In: *ICCS 2009: 9th Int. Conf. on Computational Science*. Ed. by G. Allen et al. Berlin, Heidelberg: Springer, 2009. ISBN: 978-3-642-01970-8.

[224] Magdalena Rafecas, Brygida Mosler, et al. “Use of a Monte Carlo-Based Probability Matrix for 3-D Iterative Reconstruction of MADPET-II Data”. In: *IEEE Trans. on Nuclear Science* 51.5 (2004).

[225] R.L. Graham, E.L. Lawler, et al. “Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey”. In: *Discrete Optimization II*. Ed. by P.L. Hammer, E.L. Johnson, and B.H. Korte. Vol. 5. Annals of Discrete Mathematics Supplement C. Elsevier, 1979, pp. 287–326.

[226] Nidhi Rajak, Anurag Dixit, and Ranjit Rajak. “Classification of List Task Scheduling Algorithms: A Short Review Paper”. In: *Journal of Industrial and Intelligent Information* 2 (Jan. 2014).

[227] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[228] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411.

[229] M. L. Dertouzos and A. K. Mok. “Multiprocessor online scheduling of hard-real-time tasks”. In: *IEEE Transactions on Software Engineering* 15.12 (Dec. 1989), pp. 1497–1506. ISSN: 2326-3881.

[230] Yuming Xu, Kenli Li, et al. “A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues”. In: *Information Sciences* 270 (2014), pp. 255–287. ISSN: 0020-0255.

[231] Henan Zhao and Rizos Sakellariou. “An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm”. In: *Euro-Par 2003 Parallel Processing*. Ed. by Harald Kosch, László Böszörményi, and Hermann Hellwagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 189–194. ISBN: 978-3-540-45209-6.

[232] Jong-Kook Kim, Sameer Shivle, et al. “Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment”. In: *Journal of Parallel and Distributed Computing* 67.2 (2007), pp. 154–169. ISSN: 0743-7315.

[233] G. Kannan and S. Thamarai Selvi. “Nonpreemptive Priority (NPRP) based Job Scheduling model for virtualized grid environment”. In: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*. Vol. 4. Aug. 2010, pp. V4-377-V4–381.

[234] Risat Mahmud Pathan. “Unifying fixed- and dynamic-priority scheduling based on priority promotion and an improved ready queue management technique”. In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium* (2015), pp. 209–220.

[235] P. Dhivya., V. Sangamithra., et al. “Improving the resource utilization in grid environment using Aging Technique”. In: *2012 Third International Conference on Computing, Communication and Networking Technologies (ICCCNT’12)*. July 2012, pp. 1–5.

[236] R. Armstrong, D. Hensgen, and T. Kidd. “The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions”. In: *Heterogeneous Computing Workshop, 1998. (HCW 98) Proceedings. 1998 Seventh*. Mar. 1998, pp. 79–87.

[237] Oscar H. Ibarra and Chul E. Kim. “Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors”. In: *J. ACM* 24.2 (Apr. 1977), pp. 280–289. ISSN: 0004-5411.

[238] Saeed Parsa and Reza Entezari-Maleki. “RASA: A new task scheduling algorithm in grid environment”. In: *World Applied Sciences Journal* 7 (Jan. 2009), pp. 152–160.

[239] Min-You Wu and Wei Shu. “A high-performance mapping algorithm for heterogeneous computing systems”. In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. Apr. 2001, 6 pp.-.

[240] Yuebin Bai, Cong Xu, and Zhi Li. “Task-Aware Based Co-Scheduling for Virtual Machine System”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC ’10. Sierre, Switzerland: Association for Computing Machinery, 2010, pp. 181–188. ISBN: 9781605586397.

[241] Orathai Sukwong and Hyong S. Kim. “Is Co-Scheduling Too Expensive for SMP VMs?” In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys ’11. Salzburg, Austria: Association for Computing Machinery, 2011, pp. 257–272. ISBN: 9781450306348.

[242] L. He, H. Zhu, and S. A. Jarvis. “Developing Graph-Based Co-Scheduling Algorithms on Multicore Computers”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.6 (2016), pp. 1617–1632.

[243] Major Bhadauria and Sally A. McKee. “An Approach to Resource-Aware Co-Scheduling for CMPs”. In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS ’10. Tsukuba, Ibaraki, Japan: Association for Computing Machinery, 2010, pp. 189–199. ISBN: 9781450300186.

[244] E. Frachtenberg, G. Feitelson, et al. “Adaptive parallel job scheduling with flexible coscheduling”. In: *IEEE Transactions on Parallel and Distributed Systems* 16.11 (2005), pp. 1066–1077.

[245] Y. Jiang, X. Shen, et al. “Analysis and approximation of optimal co-scheduling on Chip Multiprocessors”. In: *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 220–229.

[246] Kai Tian, Yunlian Jiang, and Xipeng Shen. “A Study on Optimally Co-Scheduling Jobs of Different Lengths on Chip Multiprocessors”. In: *Proceedings of the 6th ACM Conference on Computing Frontiers*. CF ’09. Ischia, Italy: Association for Computing Machinery, 2009, pp. 41–50. ISBN: 9781605584133.

[247] Terry R Jones, Pythagoras C Watson, et al. “Parallel-aware, dedicated job co-scheduling within/across symmetric multiprocessing nodes”. In: (Oct. 2010).

[248] Cédric Augonnet, Olivier Aumage, et al. *StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators*. Research Report RR-8538. INRIA, May 2014. URL: https://hal.inria.fr/hal-00992208.

[249] Q. Meng, A. Humphrey, and M. Berzins. “The uintah framework: a unified heterogeneous task scheduling and runtime system”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012, pp. 2441–2448.

[250] B. Peterson. “Portable and Performant GPU/Heterogeneous Asynchronous Many-task Runtime System. Ph.D. Dissertation”. PhD thesis. University of Utah, School of Computing, Dec. 2019. URL: http://www.sci.utah.edu/publications/Pet2019a/bradpeterson-thesis.pdf.

[251] Christopher J. Rossbach, Yuan Yu, et al. “Dandelion: A Compiler and Runtime for Heterogeneous Systems”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 49–68. ISBN: 9781450323888.

[252] Thomas Brandes, Eric Schricker, and Thomas Soddemann. “The LAMA Approach for Writing Portable Applications on Heterogenous Architectures”. In: *Scientific Computing and Algorithms in Industrial Simulations: Projects and Products of Fraunhofer SCAI*. Ed. by Michael Griebel, Anton Schüller, and Marc Alexander Schweitzer. Cham: Springer International Publishing, 2017, pp. 181–198. ISBN: 978-3-319-62458-7.

[253] D. K. Newsom, O. Serres, et al. “Energy Efficient Job Co-scheduling for High-Performance Parallel Computing Clusters”. In: *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*. 2015, pp. 550–556.

[254] C. Reaño, F. Silla, et al. “Intra-Node Memory Safe GPU Co-Scheduling”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (May 2018), pp. 1089–1102. ISSN: 1045-9219.

[255] Víctor J. Jiménez, Lluís Vilanova, et al. “Predictive Runtime Code Scheduling for Heterogeneous Architectures”. In: *High Performance Embedded Architectures and Compilers*. Ed. by André Seznec, Joel Emer, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 19–33.

[256] Andrew Burkimsher. “Grid Scheduling of Dependent Task Sets: A Literature Survey”. In: *EngD Literature Survey, Department of Computer Science, University of York, YO10 5DD, UK* (2010).

[257] Peter Koch. *Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors.* 1995.

[258] H. Orsila, T. Kangas, et al. “Parameterizing Simulated Annealing for Distributing Task Graphs on Multiprocessor SoCs”. In: *System-on-Chip, 2006. International Symposium on*. 2006, pp. 1–4.

[259] Thomas Becker. “Iteratives Scheduling von bedingten Task-Graphen in einem heterogenen System”. MA thesis. Karlsruhe Institute of Technology, Mar. 2014.