Karlsruhe Institute of Technology

Resource Allocation for Software Pipelines in Many-core Systems

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurswissenschaften

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

genehmigte

Dissertation

von

Janmartin Jahn

Tag der mündlichen Prüfung:	13.02.2014
Erster Gutachter:	Prof. Dr. Jörg Henkel, Karlsruhe Institute of Technology
Zweiter Gutachter:	Prof. Dr. Nikil Dutt, University of California, Irvine

Janmartin Jahn
Briegerstr. 12c
76139 Karlsruhe

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben haben und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen - die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Janmartin Jahn

Acknowledgements

I want to sincerely thank my advisor, Prof. Dr. Jörg Henkel. He shaped my research interests, challenged me, and gave me complete freedom to explore my own directions. His vision provided a great opportunity for exceptional professional and personal growth. I am very grateful for his guidance. Furthermore, I want to deeply thank my co-advisor, Prof. Dr. Nikil Dutt, University of California, Irvine. During my visit at UC Irvine, he welcomed me with great hospitality. Ever since then, his guidance, mentorship and his full support for all my endeavors has become invaluable for me.

My sincere thankfulness goes to Prof. Dr. Jian-Jia Chen for his great advice and excellent guidance. I am deeply thankful for his help, he shaped my research as well as my personal development.

My colleague, co-author, and friend Santiago Pagani contributed greatly to my success. The fruitful discussions during the long hours in which we developed ideas, algorithms, and solutions, are invaluable.

Furthermore, I want to express my deep gratitude to my colleagues and friends Hussam Amrouch, Thomas Ebi, Artjom Grudnitsky, Sebastian Kobbe, Volker Wenzel, and the other members of the Chair for Embedded Systems. I am deeply thankful for their support, friendship, and advice.

My friend Prof. Dr. Mohammad Abdullah Al Faruque helped me to start my scientific work. Through his guidance, he shared highly valuable experiences and taught me the skills essential for successful research.

Furthermore, Dr. Lars Bauer and Dr. Mohammad Shafique often contributed great advice and feedback to my work.

I want to dedicate this thesis to my parents. They always support all my endeavors and their faith in me gives me strength and fortitude.

List of Figures

List of Tables

List of Algorithms

List of Code Listings

List of Abbreviations

API	Application Programming Interface
ARB	Architecture Review Board
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
DMA	Direct Memory Access
DVFS	Dynamic Voltage and Frequency Scaling
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
IPC	Inter-Process Communication
ITRS	International Technology Roadmap for Semiconductors
KPN	Kahn Process Network
LUT	Look Up Table
MIMD	Multiple Instructions, Multiple Data
MPSoC	Multi-processor System-on-Chip
MPB	Message Passing Buffer
MPI	Message Passing Interface
NoC	Network-on-Chip
RCCE	Rocky, Intel's Communication Library for the SCC

SCC Single-Chip Cloud Computer

SDF Synchronous Data Flow

SIMD Single Instruction, Multiple Data

TDP Thermal Design Power

TFLOP Tera Floating Point Operations

Contents

List of own publications included in this thesis

Transactions (blind peer-reviewed)

[50] Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen, Jörg Henkel. “Runtime Resource Allocation for Software Pipelines”, ACM Transactions on Parallel Computing (TOPC). Submitted for review.

Workshops (blind peer-reviewed)

[48] Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, Jörg Henkel. “Runtime Resource Allocation for Software Pipelines”. ACM/EDAA 16th International Workshop on Software and Compilers for Embedded Systems. pp 96-99. St. Goar, Germany 2013.

[49] Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, Jörg Henkel. “Work in Progress: Malleable Software Pipelines for Efficient Many-core System Utilization”, Proceedings of the 6th Intel Many-core Applications Research Community (MARC) Symposium. pp 30-33. Toulouse, France 2012.

Ph.D Forum

Janmartin Jahn and Jörg Henkel. "Runtime Resource Allocation for Many-Core Architectures". ACM/SIGDA Ph.D. Forum at the IEEE/ACM Design Automation Conference (DAC), Austin, TX, USA 2013.

Conferences (double-blind peer-reviewed)

[53] Janmartin Jahn, Mohammed Abdullah Al Faruque, Jörg Henkel. "CARAT: Context-aware, Runtime Adaptive Task Migration for Multi Core Architectures", IEEE/ACM 14^{th} Design Automation and Test in Europe Conference (DATE). pp. 515-520. Grenoble, France 2011.

[52] Janmartin Jahn, Jörg Henkel. "Pipelets: Self-organizing Software Pipelines for Many-Core Architectures", IEEE/ACM 16^{th} Design Automation and Test in Europe Conference (DATE). pp. 1530-1521. Grenoble, France 2013.

[55] Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen, Jörg Henkel. "Optimizations for Configuring and Mapping Software Pipelines in Many-Core Systems", IEEE/ACM 50^{th} Design Automation Conference (DAC), Austin, Texas, USA 2013.

[54] Janmartin Jahn, Santiago Pagani, Jian-Jia Chen, Jörg Henkel. "MOMA: Mapping of Memory-intensive Software-pipelined Applications for Systems with Multiple Memory Controllers", IEEE/ACM 32^{nd} International Conference on Computer-Aided Design (ICCAD), San Jose, California, USA 2013.

Abstract

Many-core systems integrate a growing number of cores on a single chip and are expected to integrate hundreds of cores soon, while thousand-core systems are on the horizon. Despite their massive processing power, it is crucial to employ their parallel resources efficiently to benefit greatly from parallel processing. This is largely determined by *resource allocation*, i.e. by how applications are allocated to the cores.

Efficiently using massively parallel resources demands for applications with a high degree of parallelism, such as software pipelines. Software pipelines are well-suited for many-core systems as they do not require globally shared memory and they are applicable to a large class of applications, such as complex stream-processing and multi-media applications.

However, efficiently employing many-core systems is challenging as their performance is affected by many factors, e.g. by inter-task communication and by the balance of computational load among cores. A further great challenge is posed by the saturation of memory controllers that may result from unfavorably allocating memory-intensive tasks, and by unreliable hardware. Additionally, the large number of cores and their great computational power now permit highly complex application scenarios that may result in significantly varying resource requirements at runtime. In such scenarios, adapting resource allocations can be necessary to maintain a high performance. However, the state-of-the-art methods for multi-core systems cannot sufficiently address these challenges for several reasons. These reasons include a large algorithmic complexity that can lead to scalability issues for large systems, as well as architectural assumptions (e.g. globally shared memory). As a consequence, it is of crucial importance to develop novel methods for resource allocation that are well-suited for large many-core systems.

This thesis addresses these challenges by proposing novel concepts, methods and mechanisms for resource allocation in many-core systems. Based on a model of the throughput of software-pipelined applications, resources are allocated and re-allocated at runtime based on the observed system state and on the observed resource requirements of tasks. To achieve this, a number of algorithms have been developed and implemented that address the following issues:

- Calculation of optimal resource allocations at runtime.
- Scalability for very large system sizes by trading the optimum.
- Joint optimization for the computational requirements of tasks, for inter-task communication, and for avoiding the saturation of memory controllers.
- Resilience of the resource allocation method against unreliable cores.
- Efficient task migration to re-allocate resources at runtime.

This thesis divides the problem of resource allocation into sub-problems that can be solved separately. The proposed methods build upon a novel algorithm for optimally *fusing* pipeline stages, i.e. reducing the degree of parallelism of an application so that it runs on a given number of cores. A centralized, optimal method is proposed to allocate cores to applications, and is extended in a distributed, hierarchical manner to trade the optimum for a high level of scalability. Then, it is shown how the saturation of memory controllers can be avoided whenever possible, or minimized otherwise. Furthermore, this thesis proposes to alternatively allocate resources in a self-organizing way. Self-organization can be used to increase the resilience of the resource allocation methods against unreliable hardware.

The proposed mechanism for task migration builds upon an analysis of the memory access behavior of stream-processing application and employs runtime-adaptive migration policies. Furthermore, it exploits the temporal pattern of software pipelines to reduce the performance overhead of task migration so that frequent re-allocations of resources are feasible at runtime.

The resulting system has been implemented on Intel's 48-core Single-Chip Cloud Computer (SCC) and in a high-level many-core system simulator that allows to simulate arbitrary system sizes. Multiple complex real-world applications such as a H.264 video encoder and an embedded application for stereo vision-based object tracking have been parallelized to form software pipelines. In extensive experiments, significant improvements of the proposed methods upon the state of the art can be observed.

Zusammenfassung und Übersicht der Arbeit

Many-Core Systeme integrieren eine Vielzahl von Rechenkernen auf einem Chip, um Geschwindigkeitsvorteile durch die parallele Ausführung von Berechnungen zu erzielen. Solche Systeme werden als vielversprechende Möglichkeit erachtet, die Rechenleistung entsprechend dem Moore‘schen Gesetzt über die physikalischen und ökonomischen Grenzen von Ein- und Multi-Core Systemen hinaus weiter zu steigern. Es wird erwartet, dass Many-Core Systeme bald hunderte bis tausende von Kernen integrieren. Es ist von entscheidender Wichtigkeit, diese parallelen Rechenressourcen effizient zu nutzen, damit Many-core Systeme erfolgreich eingesetzt werden können.

Um in solchen Systemen die parallelen Rechenressourcen auslasten zu können, müssen die eingesetzten Anwendungen einen hohen Grad an Parallelität aufweisen. Eine wichtige Klasse solch paralleler Anwendungen stellen sogenannte Software Pipelines dar. Software Pipelines bestehen aus mehreren Stufen, die wiederholt komplexe Berechnungen auf einem Strom von Eingabedaten ausführen, wobei die Ausgabedaten einer Stufe die Eingabedaten ihres direkten Nachfolgers darstellen. Dieses Paradigma erlaubt es, jede Stufe einem einzelnen Kern zuzuordnen, somit also einen hohen Grad an Parallelität zu erreichen. Zudem können Software Pipelines durch die explizite Kommunikation in Systemen ohne gemeinsamen Speicher und ohne Cache-Kohärenz verwendet werden, was der Abkehr von diesen Konzepten aus Skalierbarkeitsgründen entgegen kommt. Software Pipelines sind insbesondere für viele Multimediaanwendungen geeignet.

Für die effiziente Auslastung der Rechenressourcen von Many-Core Systemen ist es jedoch von entscheidender Wichtigkeit, dass die parallelen Anwendungen auf eine Weise den Kernen zugeordnet werden, die den Durch-

satz des Systems maximiert. Allerdings sind herkömmliche Konzepte zur Ablaufplanung (Scheduling) nicht geeignet um die effiziente Zuordnung von Anwendungen zu Kernen zu erzielen (im Folgenden schlicht "Zuordnung" genannt). Ein Hauptgrund hierfür ist, die Frage, welche Anwendung welchen Kernen zuzuordnen ist, häufig nicht zentraler Gegenstand von Ablaufplanung in Multi-Core Systemen ist. Der Stand der Kunst, eine solche Zuordnung für Multi-Core Systeme zu erreichen, setzt meist kohärente Caches und gemeinsamen Speicher voraus und/oder ist aufgrund hoher Rechenkomplexität nicht für Systeme mit hunderten Kernen geeignet. Infolgedessen ist es von großer Wichtigkeit, neue Konzepte zu entwickeln, die eine solche Zuordnung und somit eine effiziente Nutzung der Kerne in Many-Core Systemen gewährleistet.

Die Schwierigkeiten und Hürden für solche neuen Konzepte sind allerdings vielfältig. Einerseits ist es in Many-Core Systemen, in denen viele Anwendungen gleichzeitig ausgeführt werden, häufig der Fall, dass sich der Ressourcenbedarf der Anwendungen während der Laufzeit ändert. Solche Änderungen werden beispielsweise durch Benutzereingaben oder durch Änderungen in den Eingabedaten hervorgerufen und sind oft nicht vorhersehbar. Deshalb muss die Zuordnung von Anwendungen zu Kernen gegebenenfalls zur Laufzeit geändert (und daher neu berechnet) werden. Zudem stellt die Änderung von Zuordnungen zur Laufzeit entsprechende Anforderungen an die maximale Rechenkomplexität und an die Menge der zur Berechnung benötigten Informationen.

Eine weitere Herausforderung stellt der mögliche Ausfall von Kernen dar, da es bei einer sehr großen Zahl von Kernen in Zukunft nicht mehr sinnvoll ist, die Stabilität aller Kerne zu jeder Zeit zu garantieren. Falls auf ausfallenden Kernen Instanzen des Verwaltungssystems ausgeführt werden, kann der Durchsatz des Gesamtsystems erheblich beeinträchtigt werden. Um die beschriebenen Probleme zu lösen, stellt diese Arbeit Methoden, Konzepte und Implementierungen vor, die es erlauben, effektive Zuordnungen in Many-Core Systemen effizient zu erreichen. Die Grundlage bildet die Modellierung zur Berechnung und Vorhersage des Durchsatzes von Software-Pipelines. Um ihre Zuordnungen zur Laufzeit zu berechnen und anzupassen werden mehrere Algorithmen entworfen und implementiert, die folgende Fragestellungen beantworten:

- Migration von Anwendungen zwischen Kernen, damit Zuordnungen zur Laufzeit geändert werden können.
- Berechnung optimaler Zuordnungen zur Laufzeit.

- Skalierbarkeit für beliebige Systemgrößen unter Abwägung der Optimalität der Lösung.
- Gleichzeitige Berücksichtigung von Rechenbedarf, Kommunikationsvolumen und der Auslastung gemeinsam genutzter Speichercontroller.
- Robustheit des Gesamtsystems gegen den unvorhersehbaren Ausfall von Kernen zur Laufzeit.

Das resultierende Gesamtsystem wurde auf dem Intel Single-Chip Cloud Computer, einem 48-Kern System, und in einem Simulator implementiert, der beliebige Systemgrößen simuliert. Hierfür wurden mehrere komplexe Anwendungen wie unter anderem ein H.264 Video Encoder und eine Anwendung zur sichtbasierten Objektverfolgung zu Software Pipelines parallelisiert.

Im Vergleich zum Stand der Kunst können deutliche Leistungsgewinne erzielt werden, wobei der Durchsatz des Gesamtsystems deutlich gesteigert werden kann. Zudem sind die in dieser Dissertation vorgestellten Methoden und Algorithmen auch für große Systemgrössen bis über 1.000 Kernen effektiv und effizient einsetzbar.

Chapter 1

Introduction

Many-core systems comprise hundreds of cores on a single chip to benefit from the parallel execution of tasks, and systems that contain a thousand cores are on the horizon [17]. Their massively parallel computational resources are widely regarded as a powerful means to increase the performance of computing systems through parallel processing. In the wake of their great capabilities, highly complex and computationally demanding applications, such as real-time video encoding, live object- and face recognition, on-demand data encryption, and many more, are now possible and increasingly common even in embedded systems [104].

This high degree of parallelism has emerged to overcome the limited performance of single- and multi-core systems. Due to their high levels of power consumption and greatly increasing design and verification complexity, a further scaling of core frequencies, integration sizes, and core complexity has hit its limits [17, 116]. As these are the main drivers that allowed to improve the computational power of single- and multi-core systems in line with Moore's law for multiple decades, significant improvements in the future are unlikely in many scenarios [16, 116].

A viable solution to increase the performance of a system beyond these limits within a reasonable power envelope is to integrate a growing number of simple cores on a single chip, paving the way for current and future many-core systems.

1.1 Many-core Systems: Status Quo and Trends

Many-core systems and multi-core systems are fundamentally different as large numbers of cores necessitate a departure from established paradigms, e.g. regarding the on-chip communication infrastructure and memories. This section highlights some of the most prominent architectural differences and provides a demarcation of many- and multi-core systems before giving an overview of existing many-core systems and the most important trends.

State-of-the-art many-core systems often deploy a Network-on-Chip (NoC) to enable the transfer of data between their individual cores, as NoCs provide a scalable and fast communication infrastructure. Network-on-Chips replace the bus-based interconnects that are predominantly found in single- and multi-core systems because NoCs can avoid congestion when multiple cores communicate concurrently [32], which occurs commonly in many-core systems due to their large number of cores. Furthermore, NoCs do not suffer from the excessive area overhead of global wiring.

In systems that contain off-chip memory, the Network-on-Chip may also connect the cores to one or more memory controllers that provide access to off-chip memory. As the performance of off-chip memory is often much lower than the performance of the cores, single-/multi-core as well as many-core systems often employ caches to bridge this gap. In multi-core systems, all caches are commonly *coherent*, i.e. all caches provide an identical view on the data at any time. However, as the overhead of cache coherence protocols grows with a growing number of cores, it is likely that many-core systems depart from cache coherence and, as a consequence, from the concept of sharing memory globally among all cores. As an example, Intel's Single-Chip Cloud Computer (SCC) does not employ any form of hardware cache coherence [43] and sharing memory among cores requires to disable

	Many-core Systems	Multi-core Systems
Number of cores	> 32	2,4,6,8,12,...
Communication	Network-on-Chip	Bus-based (and extensions, e.g. Intel QPI, HyperTransport)
Memory	Private	Shared
Cache	Not coherent	Coherent

Table 1.1: Demarcation: typical properties of many- and multi-core systems

caching [78]. Future systems may also follow this paradigm as the number of cores that can be integrated on a single chip is predicted to double roughly each 18 months [47], following Moore's law of transistor count doubling in each generation of processors [84].

Table 1.1 summarizes the typical architectural properties of many- and multi-core systems. While there exists no common demarcation for many- and multi-core systems, this thesis considers systems with more than 32 cores, private memories, and incoherent caches as *many-core systems*. In contrast, a system with less cores, shared memories, and coherent caches is considered a *multi-core system*. In the scope of this thesis, Graphics Processing Units (GPUs) are not considered a many-core system, even though they share many characteristics. However, GPUs are commonly employed to accelerate a specific class of applications and face different challenges than the many-core systems that this thesis focuses on.

Various many-core systems have been released in the recent past, and it is predicted that the trend to integrate more and more cores on a single chip continues [17, 47, 114]. Figure 1.1 shows the core counts of some systems. Following the curve of the International Technology Roadmap for Semiconductors (ITRS) prediction of core count doubling each 18 months, chips with hundreds and even with a thousand cores are to be expected soon.

The many-core systems that are available today include Intel's 80-core *Tera-scale Research Chip* (also known as *Polaris*) [114], and the Intel *Single-Chip Cloud Computer* (SCC) [43]. The purpose of the Tera-scale Research Chip is to achieve high computational power to prove the effectiveness of the many-core system paradigm by achieving 1.27 TFLOP/s (Tera Floating Point Operations per second) [114]. The focus of its successor, the SCC, is on the Network-on-Chip interconnection between the individual cores rather than on computing power [78]. It entirely departs from cache coherence and shared off-chip memory can only be used without data caching. Intel's Xeon Phi [31], which was released commercially in 2013, integrates 61 cores on a single chip. Tilera's TILE-Gx systems comprise up to 72 cores at 1.2 GHz. For embedded streaming multimedia applications, Toshiba's single-chip many-core system with 64 cores aims at high performance and low power consumption to allow 1080p 30fps H.264 decoding with a power consumption of approx. 400mW [82]. All of these many-core systems use a Network-on-Chip for on-chip communication. Figure 1.1 illustrates the numbers of cores of these and some other systems, as well as the year of their release.

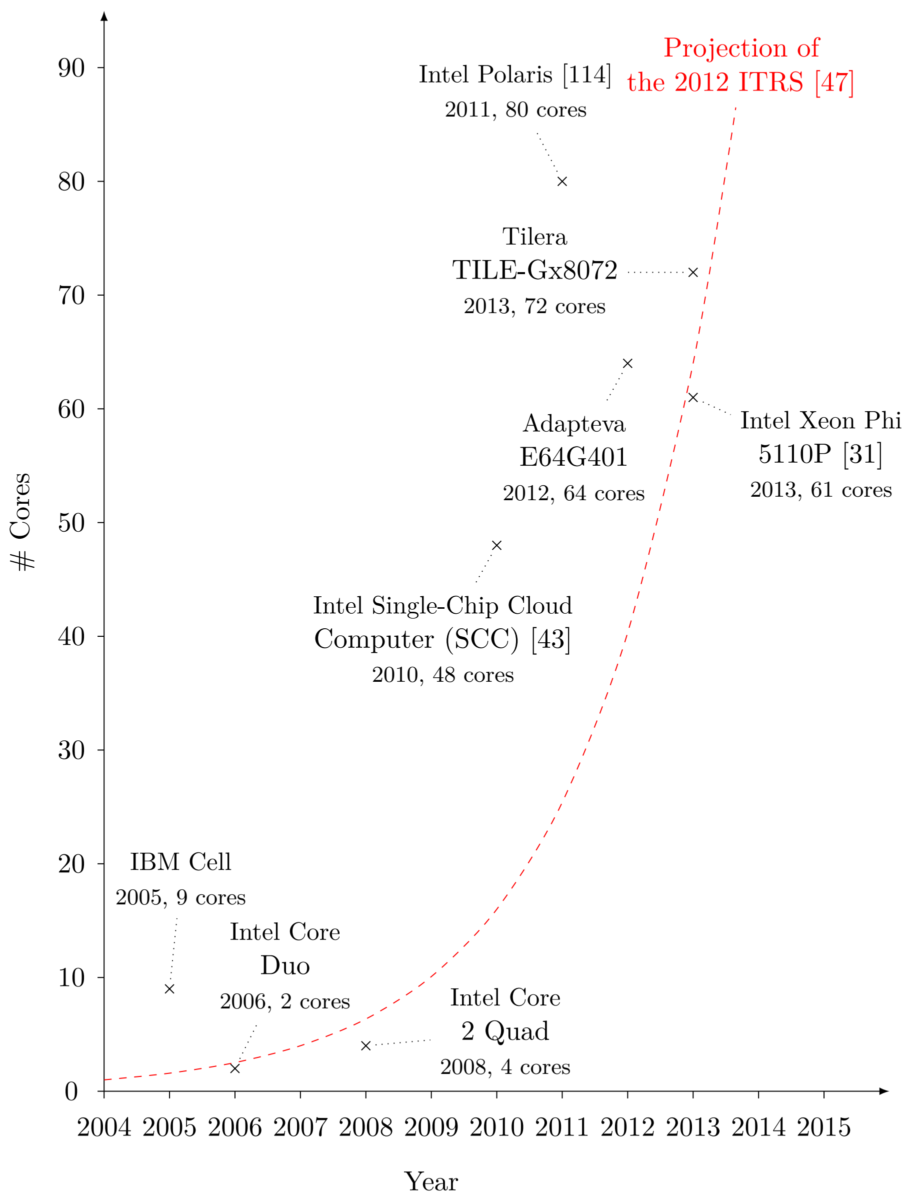


Figure 1.1: Comparison of the number of cores of selected systems and the trend in growing core counts predicted by the International Technology Roadmap for Semiconductors (ITRS) [47].

1.2 Key Challenges

The architectural differences of many-core- and single-/multi-core systems lead to a multitude of research challenges in order to efficiently employ the available computational resources. This section discusses the research challenges that are most important to successfully deploy many-core systems.

Applications should exhibit a high degree of parallelism so that a large number of cores can be used. In systems where cores may not share their memory, parallel applications must communicate shared data explicitly between their individual tasks so that many cores can be used. This is challenging as many established parallel programming concepts, e.g. OpenMP [92], POSIX Threads, and Fork/Join parallelism (see Sections 2.4.6 and 2.4.7), assume that memory is shared among all tasks and do not require the programmer to explicitly identify data that may be shared among tasks. The explicit communication of shared data that is required in distributed-memory systems can hardly be performed automatically as identifying such data is NP-hard or even undecidable in many cases [42, 67]. Furthermore, as data communication induces a performance overhead, the granularity of parallelism must be sufficiently coarse so that the communication overhead does not exceed the performance gains of parallel processing.

A parallel programming concept that is well-suited for many-core systems is *software pipelining*. A *software pipeline* is a parallel application that performs complex operations on a stream of input data. It comprises *stages* that form its individual tasks, and the output of one stage is the input of its direct successor (see Section 2.4.8 for a definition of software pipelines). The granularity of parallelism of software pipelines is coarse compared to fine-grain shared-memory parallel programming concepts such as OpenMP [92]. Software pipelines can be deployed for a wide range of applications, most prominently for complex multimedia and stream-processing applications. In the following, this thesis focuses on many-core systems that exclusively deploy software-pipelined applications.

A key challenge to efficiently employ the cores of a many-core system is the allocation of resources, i.e. the allocation of tasks to cores. Efficient resource allocation is challenging because the performance of a system is largely determined by multiple inter-dependent factors, such as the computational load of cores and on-chip communication. The impact of these factors on the system performance depends on the individual resource allocation.

The great computational power of many-cores empowers highly complex applications, such as real-time object recognition, on-the-fly video encoding, etc., and application scenarios where multiple such applications may run concurrently. This leads to *dynamic scenarios* where applications may be started, stopped, or may change their *resource requirements* (i.e. their computational requirements, their communication volumes, or their memory access behavior) at any time. Unpredictable resource requirements may necessitate the adaptation of resource allocations at runtime as otherwise, severe performance degradation could be the result. However, *resource re-*

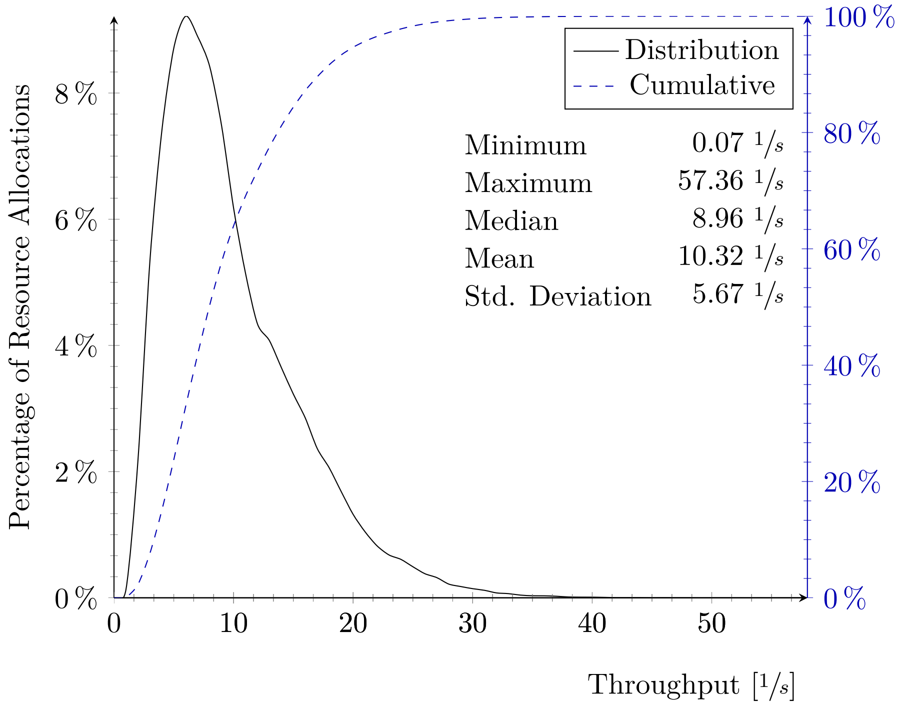


Figure 1.2: A comparison of 1,000,000 system throughputs that result from random resource allocations of the 48 cores of Intel's Single-Chip Cloud Computer (SCC) [43]. The application scenario comprises 100 communicating tasks. As less than 1% of the allocations achieve a system throughput of more than 50% (28.68 $^1/_s$) as compared to the highest system throughput observed in this experiment (57.36 $^1/_s$), this experiment illustrates the great importance of careful resource allocation.

allocation (i.e. adapting the allocation of tasks to cores at runtime) induces the overhead of *task migration*, i.e. transferring the execution of a task between cores. In many-core systems, task migration may involve to transfer the data of a task between the private memories of the corresponding cores, which can lead to a severe performance overhead.

A significant hurdle for efficient resource allocation is posed by the recent advancements towards compilers, frameworks and tools that semi-automatically aid programmers at increasing the degree of parallelism of applications, e.g. [25, 30, 95, 112, 117]. This suggests that the degree of parallelism, hence the number of tasks, may grow significantly in the future. This is challenging as the overhead of resource allocation, which depends both on the number of cores as well as on the number of tasks in a system, must remain in reasonable bounds.

Figure 1.2 illustrates the crucial importance of careful resource allocation for a high performance. The figure shows the observed system *throughput* when allocating 100 stages to 48 identical cores[1]. The *throughput* is an important metric to quantify the performance of a system. The throughput of an application is denoted by the number of *data items* (e.g. video frames, audio samples, etc.) that are processed per second. Its unit is hence $1/s$. The *system throughput* expresses the average throughput of all running applications. This thesis prefers throughput over metrics such as *makespan*, i.e. the time an applications consumes to complete its processing, because software pipelines often run perpetually. For a set of 1,000,000 different randomly chosen resource allocations, the maximum system throughput that can be observed in this experiment is 57.36 $1/s$, the minimum is 0.07 $1/s$, and more than 99.2% of the random allocations result in a system throughput of less than half of the maximum. This finding strongly highlights that resource allocation is a key challenge for the success of current and future many-core systems and must be performed very carefully. In line with these observations, resource allocation is widely regarded as one of the most important issues for many-core systems [76, 77, 106].

However, resource allocation for many-core systems faces numerous challenging hurdles:

- **Dynamism:** Unpredictable dynamism (i.e. variations in the resource requirements of tasks and the unpredictable starting/stopping of applications) can require frequent adaptions of resource allocations. Conse-

[1]This experiment uses the P54C cores of Intel's Single-Chip Cloud Computer (SCC).

quently, a system must be able to compute resource allocations quickly and it must be able to adapt allocations with minimum overhead.

- **Scalability:** The time consumed by calculating resource allocations (*computational overhead*) and for observing the relevant information (*communication overhead*) should be insignificant even for large systems and a multitude of applications.
- **Unreliable hardware:** The reliability of cores can decrease due to the continued shrinking of integration sizes beyond 22nm. This shrinking causes high power densities [86] and an increasing impact of process variations [15]. High power densities may lead to thermal problems that can cause temporary or permanent failures of cores [18, 86]. Due to process variations, manufacturing many-core systems for a guaranteed reliability may lead to low yields. Both effects suggest that in the future, systems may need to be able to operate on unreliable cores [15]. Unreliable cores can jeopardize the effectiveness of resource allocation and can thus decrease the system throughput significantly.
- **Saturation of memory controllers:** Following the rapidly increasing number of cores, bandwidth limitations of *memory controllers* (i.e. the controllers located on-chip that enable access to off-chip memory) may have a significant impact on the system throughput when running memory-intensive tasks [1]. This limitation is mostly due to the fact that both the number of pins to connect the chip with off-chip memory as well as the bandwidth of each individual pin is limited [61]. Hence, the number of memory controllers that can be integrated as well as their individual bandwidth are also limited.

  As an example, Intel's newest Xeon Phi™5110P integrates 16 memory controllers for 61 cores and each memory controller serves the accesses of approx. 4 cores [31]. However, assuming that the number of memory controllers can hardly be increased significantly due to pin count constraints [1], each memory controller may need to serve the accesses of 64 cores in a system with 1024 cores. Thus, it has to serve 16 times the requests.

  In case one memory controller serves many memory-intensive tasks, it may operate in *saturation*, i.e. it may be requested to access more data per second than it can provide [1]. This reduces the bandwidth that is available for the individual tasks, which may lead to a degraded throughput. Such a saturation of memory controllers has recently been identified as the major cause for deteriorated throughput [61]. Furthermore, when tasks communicate and they are allocated to cores that are assigned to different memory controllers, the corresponding data have to be copied from one memory controller to the other, which induces a performance penalty.

Consequently, resource allocation must account for the memory intensity of tasks so that the saturation of memory controllers can be minimized or avoided.

- **Inter-task communication:** When tasks communicate intensively, as it is the case for many complex multi-media applications, their communication overhead can be significant and may diminish the performance gains achieved through parallel processing. However, the performance penalty of inter-task communication is affected by the allocation of resources. Consequently, resource allocation should be performed in a way that optimizes for inter-task communication.

To summarize, an efficient method for resource allocation must take various resource requirements of tasks into account, it must be able to respond to their unpredictable changes in dynamic scenarios, and it should perform task migration efficiently so that a good performance can be achieved. Furthermore, it should be scalable, resilient to unreliable hardware, and should avoid the saturation of memory controllers whenever possible.

1.3 Thesis Contribution

The goal of this thesis is to achieve a high performance of many-core systems by efficiently employing their resources despite complex, unpredictable scenarios. This thesis presents novel resource allocation methods to jointly incorporate the computation, communication, and the memory access behavior of the individual tasks. By allocating resources at runtime in a highly scalable manner, allocations can be adapted quickly and thus, a high system throughput can be maintained when application scenarios or the computational requirements of the individual tasks change unpredictably. This thesis presents a cross-layer contribution and targets all aspects of resource (re-)allocation including a task migration mechanism, methods for resource allocation, and an application-level concept for software pipelines.

In particular, the novel contributions of this thesis are:

- **The novel task migration mechanism CARAT:** This runtime adaptive mechanism allows to reduce the performance penalty of task migrations which can be severe when migrations are performed frequently. CARAT aims at reducing the time consumed by task migrations by adapting the policies of transferring task data based on runtime observations. This way, tasks can be migrated quickly and with a low impact to the system throughput.

- **The novel methods for allocating resources to software pipelines CeRA, DiRA and MOMA:** Based on a model for software-pipelined applications, resources are allocated to multiple software pipelines in a way that optimizes the system throughput. This addresses two problems: Firstly, this addresses the problem of optimally allocating the available cores to a number of software-pipelined applications with different priorities that run concurrently. Secondly, the stages of each application are combined in a way that optimizes the throughput of each application, given the cores that are allocated to this application. We show how our CeRA resource allocation method can allocate resources in a globally optimal manner even when the number of stages largely exceeds the number of available cores. Furthermore, we show how our DiRA resource allocation method trades the optimum for a high degree of scalability by distributing its computations in a hierarchical manner. This way, a near-optimal throughput can be achieved for very large system sizes and numbers of tasks. As a next step, we show how our MOMA resource allocation method can jointly account for the computation of stages, their communication, and the load of memory controllers in systems with multiple memory controllers. This way, the saturation of memory controllers can be avoided whenever possible and minimized otherwise.

- **The novel concept of Pipelets:** Our Pipelets are self-organizing stages of software pipelines that interact so that resource allocations can be adapted at runtime without a controlling instance. This fully-distributed method of resource allocation allows to shift the responsibility of observing the relevant system state to the application level, which reduces the associated overhead. Additionally, self-organization allows to achieve resilient resource allocation despite unreliable hardware. Furthermore, our self-organizing software pipelines can help to restore a high system throughput when cores fail.

This thesis presents detailed analysis and descriptions of the implementations and experiments on Intel's Single-Chip Cloud Computer (SCC) [43] and in a high-level many-core system simulator. It uses complex, real-world applications including H.264 de-/encoding and stereo-vision based object recognition. The algorithms, methods and mechanisms proposed in this thesis allow to improve upon the state-of-the-art methods by modeling the throughput of software-pipelined applications and by exploiting a combination of runtime observations about the resource requirements of tasks, compile-time generated performance profiles, and application knowledge on communication patterns. This way, resources can be allocated and re-allocated at runtime in an effective and efficient manner. As a consequence, complex application scenarios can be successfully deployed to current and future many-core systems.

1.4 Outline

Before the contribution of this thesis is described in detail, Chapter 2 gives a broad overview of the background of many-core systems and Chapter 3 discusses the state-of-the-art methods for allocating their resources.

Chapter 4 presents the application model and hardware model used in this thesis and defines the problem of allocating resources. It divides the problem of resource allocation into allocating the cores to the applications (Section 4.3.1) and into allocating the cores of an application to its individual tasks (Section 4.3.2).

Chapter 5 presents our methods for allocating resources in a system-controlled way, i.e. by deploying controlling instances that are responsible for allocating the resources. Starting from our centralized, optimal CeRA resource allocation method (Section 5.2), we propose a distributed, hierarchical method DiRA that trades the optimum for a high degree of scalability (Section 5.3). Based on this, we propose MOMA to jointly optimize resource allocation for computation, inter-core communication, and for balancing the load of memory controllers in systems with multiple memory controllers (Section 5.4).

Chapter 6 proposes our concept of Pipelets for self-organizing resource allocation. The stages of our self-organizing software pipelines interact in order to allocate resources without any controlling instances (Section 6.2.1). In a next step, we show how self-organization can be combined with the system-controlled DiRA method for resource allocation in order to combine the benefits of both approaches (Section 6.3).

Chapter 7 proposes a task migration mechanism that allows to re-allocate resources at runtime, e.g. to account for unpredictably changing application scenarios or for varying resource requirements of tasks. Based on a memory access behavior analysis (Section 7.1, this thesis proposes a novel policy for migrating tasks in Section 7.3 and introduces runtime-adaptive task migration in Section 7.2 in order to account for the behavior of tasks at runtime.

Chapter 8 details the experiments and evaluations to compare the methods proposed in this thesis to the state-of-the-art methods for resource allocation. After detailing the experimental setup (Section 8.1) and the implementation on Intel's Single-Chip Cloud Computer (SCC) [43] (Section 8.2), Section 8.3 describes the many-core system simulator used for some experiments. The individual benchmark applications are described in Section 8.4 and the benchmark scenario used for our experiments is explained in Section 8.5. Section 8.6 explains how the state-of-the-art methods for resource allocations have been adapted for a fair comparison. Section 8.7 compares the system throughput that results from using the methods proposed in this thesis to the state-of-the-art resource allocation methods, while the computational overhead and the communication overhead of the proposed methods is analyzed and discussed in Sections 8.8 and 8.9, respectively.

Finally, Chapter 9 concludes this thesis and gives an outlook.

Chapter 2

Background

Many-core systems empower highly complex applications ranging from object-, face-, pattern-, and speech recognition to on-the-fly data encryption, compression, and video encoding. Furthermore, they enable novel use-cases where such tasks can be performed in real-time, and application scenarios where a multitude of such applications can run concurrently. Most importantly, they combine unprecedented computational capability with reasonable design and verification complexity and a feasible power envelope. This thesis envisions many-core systems that employ their resources efficiently by allocating and re-allocating them at runtime in a response to changing applications scenarios and changing resource requirements. Despite the focus of this thesis on resource allocation, this chapter presents a short overview of state-of-the-art many-core system architectures, memory models, and parallel programming concepts in order to provide a general background.

2.1 Many-core System Architectures

Many-core systems employ a large number of cores on a single chip to reap benefits from parallel processing. When all cores are identical, i.e. they have identical instruction sets, address width, etc., a system is commonly referred to as a *homogeneous* system. In contrast to this, systems that contain cores that are not identical are commonly referred to as *heterogeneous* systems.

2.1.1 Homogeneous Architectures

Homogeneous many-core system architectures integrate a number of identical cores. In a homogeneous many-core system, each task may be allocated to any core. Prominent homogeneous many-core systems include Intel's 80-core *Tera-scale Research Chip* (also known as *Polaris*) [114], and the *Single-Chip Cloud Computer* (SCC) [43], as well as Intel's Xeon Phi [31]. The methods for resource allocation proposed in this thesis assume homogeneous system architectures.

2.1.2 Heterogeneous Architectures

Heterogeneous systems may comprise cores with different instruction sets, hardware extensions, or power/performance characteristics, as well as specialized DSP processors or vector units. This accounts for diverging requirements of tasks: while some applications benefit greatly when data-level parallelism is exploited e.g. by SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instructions, Multiple Data) units, other applications may benefit from hardware support for video encoding, data compression, or cryptography. As a result, heterogeneous systems may deliver a higher performance and/or a lower energy consumption than homogeneous architectures [65]. However, heterogeneous system architectures are a great hurdle for efficient resource allocation as in many cases, a binary translation of program executables or retargetable code is necessary when allocating tasks to different types of cores.

As a result, the re-allocation of tasks would either be restricted by a limited set of cores to which a task can be allocated, or it could require binary translation at runtime. Both cases may induce a significant performance overhead or lead to inefficient resource allocation. Hence, the methods for resource allocation proposed by this thesis target heterogeneous system architectures.

2.1.3 Network-on-Chip (NoC)

A Network-on-Chip is an on-chip communication infrastructure that aims overcoming scalability issues by providing modular, structured, point-to-point communication [32]. In contrast to bus-based architectures or architectures where communicating components are connected via dedicated wiring, Network-on-Chips perform packet-based communication, where messages are split into *flits*. Routers transfer flits based on a routing protocol and connect the on-chip components to the Network-on-Chip. Network-on-Chips typically require a modest amount of chip area (approx. 6-7%) [32]. As they are widely regarded as a solution to the scalability issues of e.g. bus-based on-chip communication, Network-on-Chip are regarded as the common communication infrastructure for many-core systems.

2.1.4 Memory Controllers

In a many-core system, memory controllers can enable access to large off-chip memory that complements fast but small on-chip memory, such as caches or scratchpad memories. Chips are often connected to the power supply, ground, as well as to their peripherals and off-chip memory via a number of metal pins. However, this connection faces physical limitations: The total number of pins is constrained by the size of the chip packaging and the size of the individual pins. Furthermore, the data transfer bandwidth of the individual pins is constrained physically [1]. Hence, the total number of memory controllers as well as their individual bandwidth is also constrained [1].

As a result to these limitations, access to off-chip memory is often much slower than access to on-chip memory, and the access to off-chip memory via memory controllers may have a significant impact on the performance of a system [61]. To address these issues, memory controllers can optimize the memory requests by grouping them in a way that increases the throughput [61]. Furthermore, the placement of the individual memory controllers can have a significant impact on the throughput [1].

2.2 Memory Models

Multi- and many-core systems can be broadly grouped into *shared-memory systems* and into *distributed-memory systems*. In shared-memory systems, all cores share a globally accessible address space and thus, each core can access all data at any time. In contrast to this, *distributed-memory systems* employ a distributed-memory model where this is not the case.

2.2.1 Shared-memory Systems

In a shared-memory system, all cores share a globally accessible address space. Hence, tasks that are allocated to different cores may access the same data concurrently. Many paradigms for parallel programming, such as e.g. *OpenMP*, *POSIX Threads* and *Fork/Join Parallelism* (see Sections 2.4.2-2.4.7) require a shared-memory model.

The major drawback of shared-memory models is that concurrent access to one memory region may result in a bottleneck. Furthermore, they require the coherence of on-chip caches among cores. However, despite the fact that cache coherence can achieve good performance in many cases e.g. by deploying dedicated wiring [74] and by using a combination of software and hardware coherency [3], cache coherence protocols induce a significant communication overhead that commonly increases with the number of cores. Hence, many projections assume that a departure from shared-memory models is required in order to integrate hundreds or even thousands of cores on a single chip. As a result, this thesis assumes that many-core systems have distributed memories.

2.2.2 Distributed-memory Systems

As a contrast to shared-memory systems, a distributed-memory system does not allow global access to the entire address space for all cores. This ranges from cases where each core should only use private memory, as it is the case in Intel's Single-Chip Cloud Computer (SCC) [43][1], to cases where memory can be accessed in a shared manner only inside a memory island.

As a result, widely established concepts and paradigms for parallel programming, such as *OpenMP*, *POSIX Threads* and *Fork/Join Parallelism* (see Sections 2.4.2-2.4.7), may not be deployed in distributed-memory systems or face significant problems. Programming paradigms that are well-suited for distributed-memory systems include Synchronous Data Flow (SDF), Kahn Process Networks (KPN), and software pipelines.

In order to communicate data between cores, it may be necessary to transfer memory contents from the memory of the source core to the memory of the destination core. When tasks communicate extensively, this overhead

[1]On the SCC, access to shared memory requires to disable caching for all shared data, or to deploy a software-level coherence protocol.

can significantly impair the performance of a system. As this overhead depends on resource allocation, inter-core communication must be accounted for when allocating resources in such cases so that a high performance can be achieved.

2.3 Software-level On-chip Communication

In many cases, tasks need to communicate. This section discusses established concepts for application-level inter-task communication in systems with shared and distributed memories: Inter-Process Communication (IPC), Message Passing Interface (MPI), and Berkeley Sockets.

2.3.1 Shared-memory Inter-Process Communication

In a shared-memory system, two tasks can communicate efficiently using shared-memory Inter-Process Communication (IPC). This section discusses methods and mechanisms for shared-memory IPC in multitasking and multithreading operating systems that support virtual memory, such as Windows NT and many Linux and Unix-based operating systems.

The virtual address space of each task is private. Consequently, when different tasks need to communicate via shared-memory IPC, they must first gain access to a region of memory that is shared among their tasks. There are the following means to achieve this:

Named Pipes A named pipe is a persistent object that acts as a First-In, First-Out (FIFO). Usually, named pipes connect two counterparties, a producer that writes to the named pipe, and a consumer that reads from it. Due to the nature of FIFOs, named pipes are usually employed for one-way communication.

Memory-mapped Files Memory-mapped files allow two tasks to share memory. Any number of tasks can access one memory-mapped file object (e.g. by calling `CreateFileMapping`/`CreateFileMappingNuma`), which is identified by its name. Memory-mapped files may be used to communicate or store data that is larger than the virtual address space of the system permits. This can be achieved by mapping only portions of the file object into the virtual address space that do not exceed this limit (e.g. `MapViewOfFileEx`).

Remote Memory Access Another way to achieve IPC via shared memory is when one task allocates memory within the virtual address space of another task, and passes the handle to this memory location. In Windows NT-based operating systems, `VirtualAllocEx` allows to allocate memory remotely in the virtual address space of another task. However, to initially establish an IPC connection, the handle to this memory must be passed via other IPC mechanisms.

2.3.2 Message Passing

Message passing is a paradigm for inter-task communication in shared- and in distributed-memory systems. Generally speaking, message passing allows for *point-to-point* (producer/consumer) communication and *collective communication* (e.g. multicast and broadcast). A producer is required to explicitly send the data for communication to its consumer(s). A well-established, standardized implementation of this paradigm is the Message Passing Interface (MPI), implementations include e.g. OpenMPI [94]. MPI supports various integral as well as user-specified data types, and abstracts specifics of the communication infrastructure.

Message passing is well-suited for distributed computing systems and many-core systems without shared memory and cache coherence.

2.3.3 Berkeley Sockets

Berkeley sockets (the “BSD socket API”) provide an Application Programming Interface (API) for Inter-Process Communication (IPC), most commonly across computer networks. Berkeley sockets originated in 1983 as a C language API and have emerged as a de-facto standard for socket communication based on the Internet Protocol (IP) standard. Many modern operating systems, including Linux, UNIX-based Systems, and Windows (through WinSock) provide implementations of Berkeley sockets.

Berkeley sockets can be used for inter-task communication in shared- and in distributed-memory systems. Table 2.1 summarizes the most important functions of the API (Source: [33]).

Function	**Description**
socket	Creates a new socket with a specified type and reserves system resources for it.
bind	Server-side function to associate a socket with a given socket address structure to specify e.g. the local port and address.
listen	Changes a given socket so that incoming TCP connection requests will be processed and queued for acceptance.
connect	Client function to establish a connection and to assign a free local port to a given socket.
accept	Dequeues the next connection. If the queue is empty, accept blocks until the next connection request arrives.
send, sendto	Sends data to a remote socket.
recv, recvfrom	Receives data from a remote socket.
close	Closes a socket and frees reserved system resources.
gethostbyname	Allows to resolve a host name.
gethostbyaddr	Allows to resolve a host addresses.
select	Used to wait for a set of sockets to be ready for reading/writing.
poll	Checks the state of a set of sockets and can be used to determine if any socket can be written to or read from.
setsockopt	Used to set the value of a particular socket option.
getsockopt	Used to retrieve the value of a particular socket option.

Table 2.1: Summary of the most important functions of the Berkeley socket API. Similar implementations are found on many common operating systems such as Linux, Unix, and Windows. Source: [33]

2.4 Parallel Programming Concepts

In order to exploit the resources of a parallel system, applications must depart from the paradigm of sequential processing and provide parallelism on software level. This section discusses the most established state-of-the-art parallel programming concepts.

2.4.1 Terms and Definitions

The following terms will be used in the context of parallel programming.

Address Space In the context of this thesis, the *address space* of a task refers to the memory that it is assigned by the operating system. Usually, in systems that support the concept of virtual memory, the address space is private to its task.

Program A program is an abstract, passive construct characterized by a set of instructions formulated by a programmer. When executed, one or more *tasks* are the actual active instance of the program.

Task A task is an operating system construct that characterizes the executing instance of a program, containing its loaded program code. Each task contains one or more threads that execute concurrently. In systems that support the concept of virtual memory, each task is usually assigned a private memory region, i.e. its *address space*.

Thread In the context of this thesis, a *thread* is an operating-system construct that describes a sequence of program instructions that may be managed independently by the operating system scheduler. Each *task* consists of one or more threads. When a program is executed, a task and a master thread is created. This master thread may create multiple subsequent threads. All threads inside one task share its address space and thus, locking mechanisms must be employed in most instances to synchronize the access to shared memory regions.

Multithreading Multithreading is an operating system concept that allows to run multiple threads to run concurrently, and it allows each task to create more than one thread. The operating system scheduler may use techniques to time-multiplex processor resources, such as *preemptive multithreading* where each thread is assigned a time slice and is interrupted after this time slice has been exhausted. Then, another thread is assigned a time slice, and so on.

2.4.2 OpenMP

OpenMP is an Application Programming Interface (API) for shared-memory parallel programming in C/C++ and Fortran programs. It is available on many major architectures, including Unix, Linux, and Windows NT. Its Architecture Review Board members include among others AMD, Cray, HP, IBM, Intel, NEC, NVIDIA, and Texas Instruments [93]. OpenMP is characterized by a set of compiler directives and environment variables that guide the compiler to create parallel code. OpenMP achieves multithreading by distributing work to a number of concurrently operating threads that are created by the OpenMP runtime library.

In an OpenMP program, the actions taken by the compiler and by the runtime system to achieve parallelism are entirely specified by the programmer. Consequently, no data dependency analysis has to be performed by the compiler.

OpenMP supports Single Instruction, Multiple Data (SIMD) parallelism, work sharing (omp parallel for), and thread-level parallelism (omp parallel). It also provides constructs for attributing data sharing, synchronization, scheduling control, and conditional statements.

However, as OpenMP focuses on shared-memory systems, parallel OpenMP programs are not suitable for systems without shared memory and cache coherence.

2.4.3 OpenCL

The Open Computing Language (OpenCL) is an API with the main purpose of providing a unified computational abstraction framework to employ both CPUs and GPUs [60, 111]. OpenCL implements kernels in a language similar to C99, but disallows function pointers, recursions, bit fields, and arrays of variable length. However, it supports vector operations and, similar to OpenMP, it provides synchronization primitives and qualifiers to attribute the sharing of data.

OpenCL aims at a high degree of portability across platforms and thus provides an abstracted memory concept and hides specific GPU features. The memory abstraction model tightly couples data memory with its kernel, and sharing of memory is only possible via explicit data transfer. This way, OpenCL programs can be run on many-core systems without globally shared memory and cache coherence [71].

2.4.4 CUDA

The Compute Unified Device Architecture (CUDA) is an API and programming framework for parallel processing on GPUs. Similarly to OpenCL, it supports a recursion-free subset of C++ and memory spaces are private for its threads. Due to its focus to a specific set of GPUs, CUDA programs cannot run on other many-core systems.

2.4.5 Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) is a library for parallel programming. It abstracts threads to computational objects which can be allocated to cores dynamically [46]. It focuses on task graphs that are modeled by the programmer, where each computational entity is executed with respect to graph dependencies. A TBB scheduler uses a task stealing concept to balance the load among the cores of a system. Additionally, TBB provides a collection of parallel programming constructs, including parallel `for`, `reduce`, and `scan` primitives, as well as primitives for mutual exclusion and atomic operations. To allow concurrent access to shared data, TBB provides data structures for queues, vectors, and maps.

2.4.6 Threads / POSIX Threads

Running multiple *threads* (in the sense of Section 2.4.1) concurrently for one task is a concept commonly used to achieve parallelism in shared-memory systems. Multiple threads commonly share the resources (including the virtual address space) of its task but execute independently. Due to this sharing, running multiple threads generally requires shared memory and cache coherence. Extensive support for multithreading can be found in many established operating systems including Microsoft Windows and many Linux distributions. Typically, the computational resources of the cores are multiplexed by the operating system scheduler, e.g. through time-slicing.

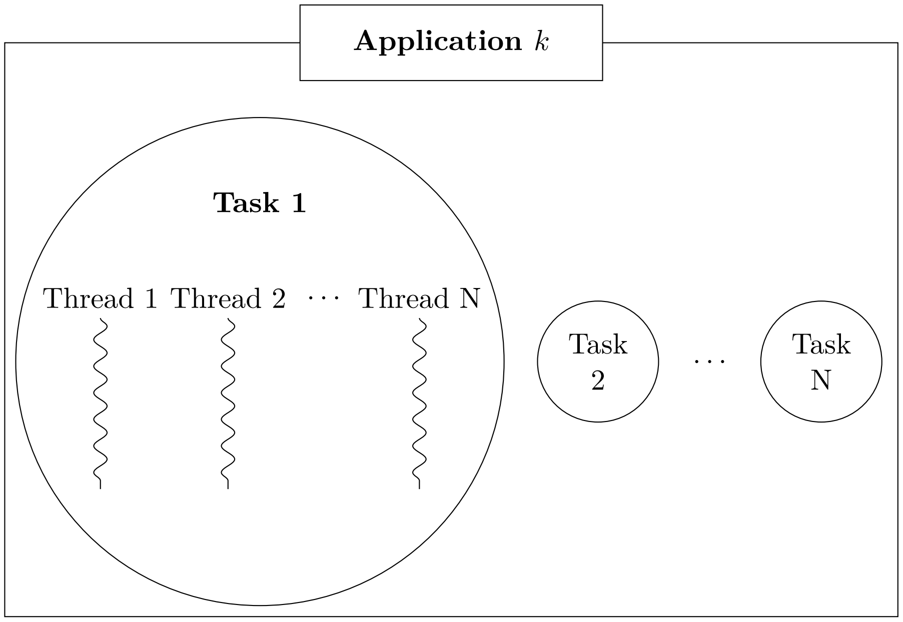


Figure 2.1: Example of multithreading: Each application contains multiple parallel task that may contain one or more concurrent threads. Each task may be allocated to a separate core, while all threads of a task must be allocated to the same core.

Such multiplexing is of vital importance when the total number of threads in a system may exceed the number of cores, which is a common scenario. Figure 2.1 shows an example where some tasks comprise multiple threads.

Threads are typically independent of each other. In order to prevent races for shared resources between threads (*race conditions*) or to synchronize behavior, the operating system must provide corresponding functions. As an example, Table 2.2 shows the thread control functions provided by Windows NT-based operating systems.

2.4.7 Fork/Join Parallelism

Fork/Join is a parallel programming concept where, at a given point of execution, a thread is split into a number N of concurrent sub-threads (*forked*). The forked sub-threads may be synchronized with a barrier (*joined*) to collect their results. Upper and lower bounds on the mean response time of the

Thread Function	**Description**
CreateRemoteThread	Creates a thread within another task.
CreateThread	Creates a new thread.
GetCurrentThread	Retrieves a handle of the current thread.
GetCurrentThreadId	Retrieves the id of the current thread.
GetExitCodeThread	Retrieves the exit code of a thread. Used to determine if a thread has finished.
GetThreadId	Get the id of a thread.
GetThreadPriority	Get the priority of a thread.
OpenThread	Opens a thread, identified by its id.
ResumeThread	Resumes a suspended thread.
SetThreadPriority	Sets the priority of a thread.
SetThreadStackGuarantee	Sets the minimum size of the stack of a thread.
SuspendThread	Suspends a thread.
SwitchToThread	Yields execution of the calling thread and gives up its remaining time slice.
Thread32First	Used to enumerate the running threads.
Thread32Next	Used to enumerate the running threads.

Table 2.2: Overview of the thread control functions of Windows NT.

sub-threads can be derived that grow proportionally with N [88]. Fork/Join parallelism is a common means to parallelize applications on Linux and Unix-based operating systems and provides an abstraction over threads.

2.4.8 Software Pipelines

In the context of this thesis, a software pipelines is a parallel program that performs complex operations a stream of input data. Its operations are divided into a set of tasks that run concurrently, i.e. into its *stages*. Each stage is a task that performs computations on a *data item*, e.g. on a video frame or audio sample. Each stage can be allocated (and re-allocated) to a core individually. The output data of one stage forms the input data of its direct successor. Parallelism is achieved by overlapping the execution of the stages, which corresponds to overlapping the processing of consecutive data items. Hence, the throughput of the application is limited by its slowest stage. Figure 2.2 illustrates a software-pipelined application that processes video frames.

Software pipelines are regarded as an effective and efficient means for parallelizing complex stream-processing applications [70, 97]. Their concept of

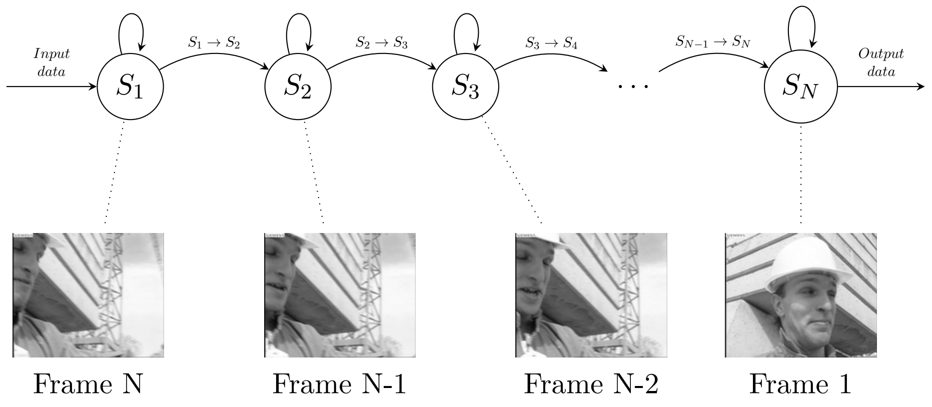


Figure 2.2: Illustration of a video-processing software pipeline with N stages and the N data items (frames 1 to N-1) that are processed in parallel. After a stage S_i finishes processing data item j, it sends its output to stage S_{i+1}, receives data from S_{i-1}, and continues with processing data item $j+1$. Stage S_{i+1} continues the processing of data item j.

overlapping the processing of different data items assumes that a program processes a large number of data items. Due to their execution model, they are inherently dead-lock free and functional determinism can be verified formally [34].

Software-pipelined programs can be created manually by the programmer, by using programming languages to explicitly express pipelined parallelism such as Brook [19], TStreams [63], Rapidmind [83], PeakStream [96], or StreamIt [115], or by using parallelizing compilers, e.g. [25, 30, 97].

2.4.9 Kahn Process Network (KPN) and Synchronous Data Flow (SDF)

Kahn Process Networks (KPN) are a model to describe parallel programs, where the individual tasks communicate via unbounded FIFO channels. Parallel programs formulated as KPNs are *deterministic* (i.e. given identical input, they produce the same output). Furthermore KPNs are *monotonic*, which means that partial output data can be derived from partial input data [58].

Synchronous Data Flows (SDF) [69] are a model for stream-processing applications similar to KPNs. SDFs are commonly used for the synthesis of sequential signal processing applications and for obtaining predictable performance, e.g. for embedded system-on-chips [37]. They form a restriction of Kahn Process Networks because their nodes produce (and consume) a fixed number of *tokens* (i.e. atomic data items that characterize the communication of two tasks for one firing) per firing.

2.4.10 Synchronization

When multiple threads access objects and resources (such as memory) that are shared among them, accesses must be synchronized to avoid concurrent access that may result in undefined behavior. The most prominent concepts that facilitate such synchronization are described in Table 2.3.

Mechanism	**Description**
Barrier	A barrier for a group of threads implies that all threads must wait at this point until all other threads reach this barrier. Examples of functions that implement this behavior are `WaitForSingleObject` and `WaitForMultipleObjects`.
Critical Section	A critical section defines a region that may be accessed exclusively by a single thread. Each thread that tries to enter a critical section must wait until another thread that already entered this critical section leaves it. Examples of functions that implement this behavior include `EnterCriticalSection`, `LeaveCriticalSection`, `TryEnterCriticalSection`, `CreateCriticalSection`, and `DestroyCriticalSection`.
Event	An event is an object that can be waited for by any number of threads. When an event is fired, all threads that wait for it are resumed. While waiting for an event, a thread is usually not scheduled for execution. Examples of functions that implement this behavior include `CreateEvent`, `OpenEvent`, `SetEvent`, and `ResetEvent`.
Lock (Semaphore)	A lock / semaphore is similar to a critical section. Their behavior differs with respect to the number of concurrent accesses: within a critical section, at most one thread may execute. For a lock, a maximum number $l \geq 1$ of concurrent lock owners may be specified.
Monitor	A monitor is an object that protects its functionality from being executed concurrently by multiple threads using critical sections. Programming is simplified by shifting the responsibility for synchronization to the synchronized object.
Mutex	See critical section.

Table 2.3: Common concepts to synchronize the execution of parallel applications.

2.5 Summary

In the domain of many-core systems, a variety of system architectures, memory models, communication paradigms and parallel programming concepts exists. However, this thesis focuses on homogeneous many-core systems with distributed memories, message passing inter-task communication, that exclusively deploy software-pipelined applications for the following reasons:

System Architecture Heterogeneous many-core systems significantly increase the complexity of resource allocation and may severely constrain the re-allocation of tasks to cores in many cases. Hence, this thesis focuses on homogeneous system architectures.

Memory Model As shared memory may potentially become a bottleneck and because they require cache coherence, future systems that integrate hundreds or even thousands of cores may potentially depart from globally shared memory in favor of distributed memories.

Inter-task Communication Message passing provides a well-established concept for on-chip communication in distributed-memory systems.

Parallel Programming Concept This thesis focuses on software-pipelined applications as they allow to parallelize a large class of complex, real-world applications, e.g. many stream-processing applications. Furthermore, their reduced expressiveness as compared to Kahn Process Network (KPN) and Synchronous Data Flow (SDF) allows for modeling the throughput of systems that run complex application scenarios.

Chapter 3

Related Work

The performance of embedded many-core systems is one of the most important concerns and has attracted a significant body of research in the recent past. This chapter discusses the state-of-the-art methods, mechanisms and strategies in this domain. It starts with an overview of important state-of-the-art methods in increasing the performance of many-core systems before detailing the state-of-the-art in resource allocation. Since adapting resource allocations at runtime may require to migrate tasks and as this implies a performance overhead that may be significant, Section 3.3 discusses the state-of-the-art mechanisms for efficient task migration.

3.1 High-performance Many-core Systems

Diverse methods exist to increase the performance of embedded many-core systems. This section highlights the most important state-of-the-art methods on architecture level, system level, and on application level.

3.1.1 Architecture-level Methods

On architecture level, domain-specific methods can significantly increase the performance of embedded many-core systems.

It was shown by [68] that the performance of stream-processing embedded Multi-processor System-on-Chips (MPSoCs) can be increased by an auto-

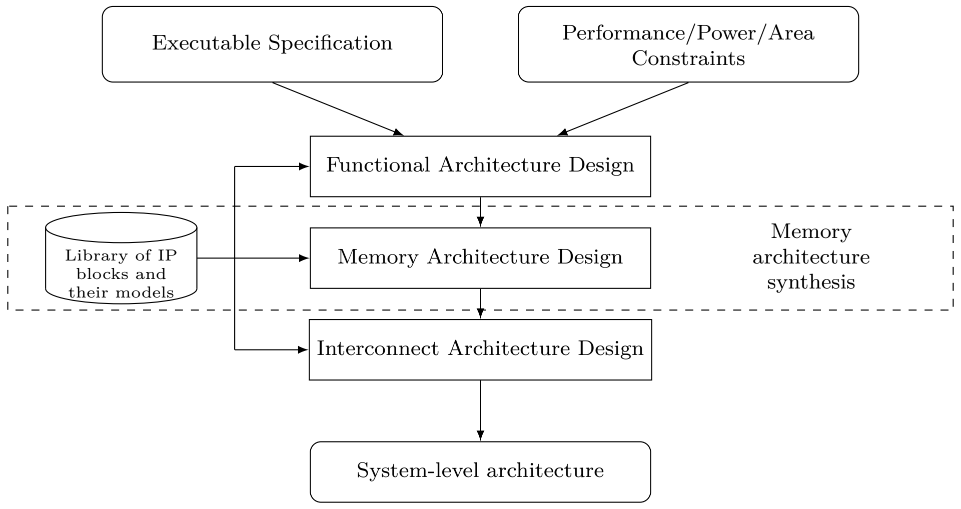
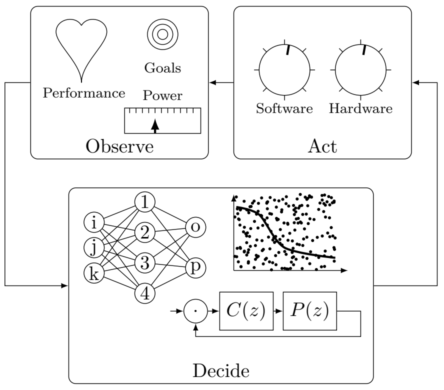

Executable Specification
Performance/Power/Area Constraints
Functional Architecture Design
Library of IP blocks and their models
Memory Architecture Design
Memory architecture synthesis
Interconnect Architecture Design
System-level architecture

Figure 3.1: Schematic overview of [68] to increase the throughput of embedded stream-processing processors by automatically synthesizing the memory architecture.

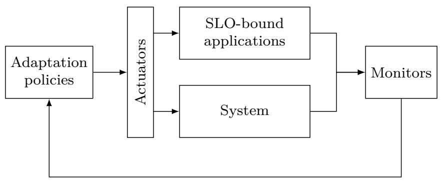
matic synthesis of the memory architecture. This method is illustrated in Figure 3.1: based on a specification of the executable and on a definition of performance, power, and area constraints, a system-level architecture is synthesized automatically. This automatic synthesis chooses the memory elements as well as NoC routers and network interfaces in a way that the performance and area constraints are satisfied, while the power consumption of the system is minimized.

Similarly, the software-hardware codesign method that is presented by [87] for FPGAs can significantly improve the performance of embedded machine-vision and object recognition applications. This method extracts domain-specific hardware accelerators automatically from an application description. It targets applications that are based on the HMAX model [105], which is a brain-inspired hierarchical model to abstract the representation of visual perception.

Furthermore, the method presented by [35] increases the performance of a system by improving the throughput of the network links that connect an embedded many-core system with 3D-stacked DRAM. This work accounts for application demands by choosing the buffer sizes of Network-on-Chip routers using a static analysis at design-time. This way, congestion in the



(a) The self-aware Observe / Decide / Act control loop of [40].



(b) Autonomic system-level control loop to maximize so-called *service-level objectives* (SLO) used by [11].

Figure 3.2: Two control-theory based methods to adjust system properties at runtime [11, 40].

most-used network links can be alleviated.

The authors of [81] have shown that the performance of many-core systems can be increased by employing multiple separate, heterogeneous Network-on-Chips on a single chip. This method exploits application knowledge as it analyzes the behavior of applications at runtime. Applications are classified into groups, e.g. into a group of applications that are more sensitive to network latency, or into a group of applications that are more sensitive to network bandwidth. Based on this classification, the data of each application is routed via one of the NoCs. This way, the performance of the system can be increased and its energy consumption can be reduced.

3.1.2 System-level Methods

A common method to achieve a high performance on system level is to employ control-theory-based methods. This way, the goal of a high performance can be combined with multiple other, possibly conflicting objectives, such as e.g. a low energy consumption. Figure 3.2 (a) illustrates the control loop used by [40]. This method proves the effectiveness of using machine learning for adjusting multiple system configuration parameters at runtime. It requires applications to specify their goals individually and it allows to react to unpredictable changes in the workload and to unpredictably changing applications scenarios. Based on these goals, the impact of changing the voltage/frequency level of cores and of changing the policy of data caches are estimated at runtime and the performance of a system can be increased by selecting the most effective configurations.

Furthermore, systems can increase their performance through a control-theory-based method to account for so-called *service-level objectives* (SLOs), which correspond to goals such as a low energy consumption or a maximum core temperature threshold. The work of [11] shows that autonomous monitoring and adaption policies can be employed effectively in a control loop as illustrated in Figure 3.2 (b).

For embedded many-core systems that combine non-volatile memories and SRAMS, runtime memory virtualization can increase the performance by adjusting memory allocation policies [12]. The authors analyze the volatility of the memory access behavior of applications in order to decide upon the physical location of data when it is allocated. It chooses from memory allocation policies that have been customized at compile time using a best-effort approach.

3.1.3 Application-level Methods

On application level, the performance of embedded many-core systems can be increased by a compiler that helps programmers in unifying the use of otherwise isolated parallel programming concepts such as OpenMP [92], OpenCL [111], etc. [5]. This way, applications can be compiled for general-purpose cores, for Graphics Processing Units (GPU), and for Field-Programmable Gate Arrays (FPGAs) without requiring to implement algorithms for each target core. Such an approach can increase the performance of a system and helps to efficiently employ heterogeneous many-core systems.

Furthermore, parallel applications may expose adjustable parameters to an application-level controller that does not require knowledge about the semantics of such parameters. Instead, a so-called auto-tuner can employ evolutionary search strategies to find good configurations of all parameters at runtime and this way, the performance of a multi-core system can be increased significantly [119].

Adjusting workloads at runtime to increase the throughput of an application while providing design-time analyzability to allow design-time performance tuning and verification can be achieved by parameterized polyhedral process networks [118]. This model of computation extends the Kahn Process

Network (KPN) model to generate different executions of each task while allowing to verify the application for deadlocks at design-time.

3.1.4 Summary of High-performance Many-core Systems

To summarize, diverse methods, mechanisms and strategies exist to increase the performance of embedded multi- and many-core systems on the architecture-, system-, and on the application level. However, the methods discussed in this section do not answer the question how parallel applications may be allocated to the cores of a system. In the following, the state-of-the-art methods for resource allocation are discussed.

3.2 Resource Allocation

Resource allocation is of paramount importance for achieving a high performance of many-core systems by efficiently employing their parallel resources. The growing computational power of many-core systems exacerbates this challenge of efficiently allocating resources as it allows more complex applications and application scenarios, which may lead to unpredictable resource requirements at runtime. This thesis envisions highly parallel many-core systems that allocate their resources efficiently through exploiting application knowledge. Such systems should adapt resource allocations when the application scenario or the resource requirements of the individual applications change unpredictably. This way, such systems can achieve and maintain a high performance at runtime.

To give a comprehensive overview, this section discusses purely design-time methods for resource allocation (i.e. once resources are allocated, no changes are made), and runtime resource allocation methods that may adapt allocations at runtime. Due to the widely acknowledged importance of resource allocation, significant work has been done in this domain. Similarly to the taxonomy presented by [106], we group the state-of-the-art resource allocation methods into design-time and runtime methods. We then further group the resource allocation methods into methods for software pipelines and parallel applications that follow a similar parallel programming paradigm, such as Synchronous Data Flow (SDF) and Kahn Process Network (KPN), and into methods for general parallel applications. As many established runtime resource allocation methods for general parallel applications assume shared

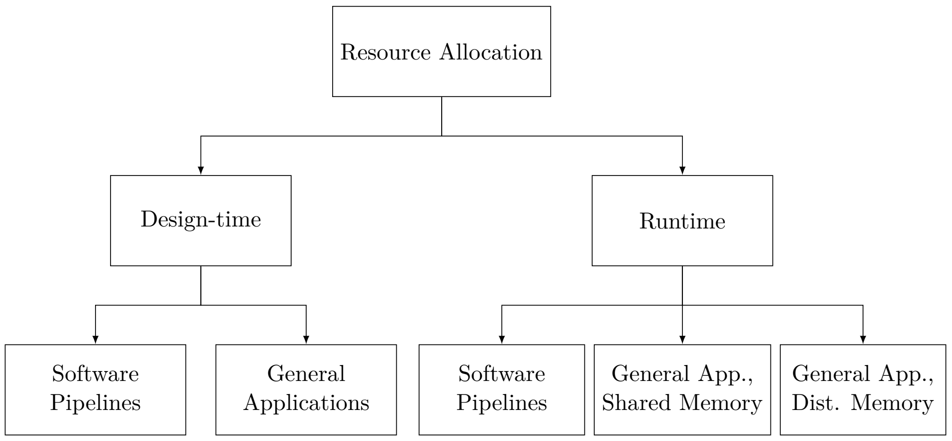


Figure 3.3: A taxonomy to group the state-of-the-art methods for allocating resources.

memory, we group them into methods for systems with shared- and with distributed memory. This taxonomy is illustrated in Figure 3.3.

3.2.1 Design-time Methods

Software Pipelines, KPN, and SDF

A scenario-based approach for resource allocations for Kahn Process Network (KPN) is proposed by [104]. At design-time, resource allocations are calculated for each possible set of concurrently running applications. At runtime, resources are allocated based on the current scenario, while the transition between scenarios is triggered by observations of the system state.

For application scenarios that consist of a single application, multiple resource allocation methods have been proposed recently. Stream-processing applications modeled as Synchronous Data Flow (SDF) can be allocated to multi-core processors using design-time adjustments of the granularity of parallelism through loop unrolling [23]. This approach employs compiler techniques that aim at maximizing the use of on-chip scratchpad memory and combines this with compiler-instrumented data buffering and background data prefetching. Similarly, SDF-based applications can be allocated to SPM-based multi-/many-cores based on an evolutionary algorithm-based technique as proposed by [26]. This method generates a static schedule at design-time, which includes a schedule for prefetching data from off-chip DRAM to on-chip scratchpads using hardware DMA units.

In order to guarantee a minimum throughput, [14] proposes a Constraint Programming-based approach to address both problems of resource allocation and scheduling simultaneously. This method calculates optimal schedules and relies on an aggressive pruning strategy to reduce its computational overhead by reducing the search-space of possible resource allocations.

For hard-real-time tasks, [9] proposes to decompose cyclo-static SDF graphs into asynchronous sets of periodic tasks with implicit deadlines. Under some conditions, this allows to achieve the maximum throughput. In order to achieve this, the authors propose to allocate resources to task subsets and focus on finding good task set representations.

In order to jointly optimize resource allocation for computation and communication, [22] proposes a heuristic design-time resource allocation phase following a phase of application synthesis of Kahn Process Networks (KPN). However, a large runtime results due to the large complexity that arises from the expressiveness of KPNs.

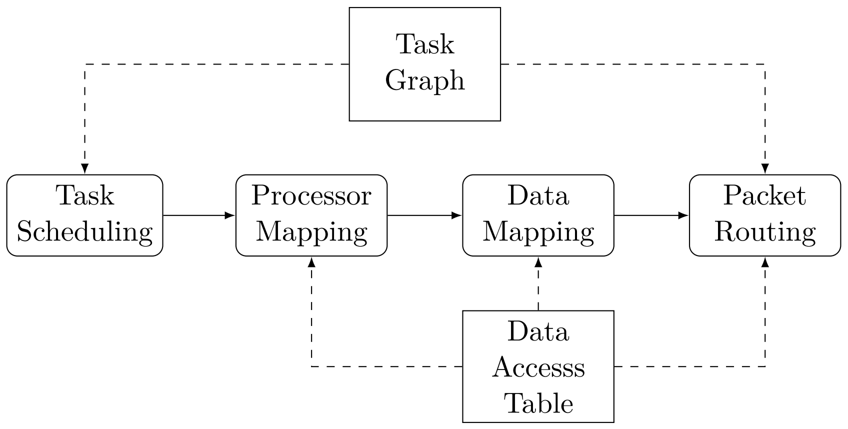


Figure 3.4: High-level overview of [24]. The rectangular boxes “Task Graph” and “Data Access Table” represent the input data, while rounded-corner boxes illustrate the phases of the approach. Task scheduling is performed independently, while scheduled tasks are later allocated to cores. In a next step, data is mapped into memories based on this processor map, and then a packet routing schedule is generated that aims at improving the performance and at reducing the energy consumption.

General Parallel Applications

A compiler-based resource allocation method to jointly optimize for communication and computation is presented by [24]. A high-level overview is illustrated in Figure 3.4. It explores the search space of resource allocations given an assumption on the communication volumes on compile time.

Similarly, [51] proposes to take into account the Network-on-Chip topology in order to reduce on-chip communication volumes. The authors focus on many-core systems with heterogeneous cores and an irregular, custom Network-on-Chip topology. To achieve a high performance, the approach formulates an exact mixed-integer quadratic programming, which has been proven to be NP-hard by [101], and then employ a genetic algorithm, a heuristic, and a successive relaxation of constraints to calculate resource allocations.

With the assumption that the number of tasks matches the number of cores, [102] uses a swarm-optimization based algorithm to minimize the on-chip communication volume by reducing hop counts between communicating tasks at design time.

A statistical approach based on extreme value theory is presented in [98]. The authors propose to generate a large random set of resource allocations and choose the best instance from this set. However, its calculations have a runtime of 25 minutes to 2 hours even for medium problem sizes.

Optimizing the allocation of resources for the memory requirements of tasks has been shown to reduce the energy consumption of a many-core system [57]. This method refines resource allocations iteratively using evolutionary algorithms at design time for satisfying a multi-objective optimization problem.

Limitations of design-time resource allocation methods

Design-time resource allocation methods aim at scenarios where the set of running applications, the resource requirements of the individual tasks, as well as their communication behavior is highly predictable at design time. However, when these assumptions are violated and unpredictable runtime scenarios do occur, design-time resource allocation methods may not be able to achieve a high system throughput. Consequently, they can hardly be applied to dynamic scenarios.

Furthermore, some design-time methods incur a high computational overhead (e.g. [98, 102]) and hence, they may not be able to deliver acceptable runtime for large-scale problems [106]. These drawbacks can be overcome by runtime resource allocation methods.

3.2.2 Runtime Methods

In dynamic scenarios, key properties of the runtime scenario are deemed unpredictable at design time. Such dynamic scenarios are increasingly common as state-of-the-art embedded many-core systems offer computational power that allows to run multiple highly complex parallel applications simultaneously. Runtime resource allocation methods can allocate the tasks to the cores of a system based on observations on the resource availability, on the set of concurrently running applications, and on the individual resource requirements of tasks. Hence, they may be able to achieve a higher performance in dynamic scenarios as compared to design-time methods. However, the computational overhead for calculating resource allocations as well as the communication overhead required for observing the system state of runtime methods is crucial as it can negatively affect the throughput of a system [106].

Consequently, it is key to allocate resources in an efficient and effective way. In the following, the state-of-the-art runtime resource allocation methods for dynamic scenarios are discussed.

Software Pipelines, KPN, and SDF

The authors of [70] propose a two-step approach, where software pipelines are compiled into retargetable executable code as illustrated in Figure 3.5. The compile-time step involves to generate a compile-time application profile. Based on this profile, the second step is a dynamic runtime scheduler which allocates the application to the cores. Its objective is to maximize the performance of the application and it considers the properties of heterogeneous cores and scratchpad memories. However, the authors assume that the resource requirements and the set of running applications does not change at runtime and thus, their approach does not adapt resource allocations at runtime. Hence, dynamic scenarios can lead to situations of significantly impaired throughput as compared to resource allocation methods that do adapt resource allocations at runtime.

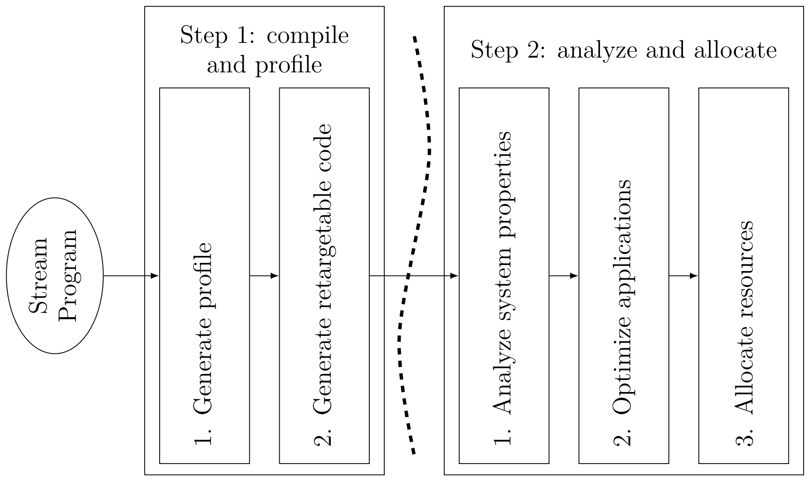

Stream Program
Step 1: compile and profile
1. Generate profile
2. Generate retargetable code
Step 2: analyze and allocate
1. Analyze system properties
2. Optimize applications
3. Allocate resources

Figure 3.5: Flow of the resource allocation method proposed in [70]. At compile time, an application profile is generated and retargetable code is created. At runtime, the application is optimized (e.g. loops are unrolled) and allocated to the resources.

A cooperative software-hardware framework to identify and accelerate tasks that limit the performance of a parallel application is presented by [56]. This approach focuses on software pipelines for shared-memory multi-core systems and shows that using hardware support, the stage that limits the throughput of a software-pipelined application can be improved by preemptively suspending others. The required hardware support for this approach includes a hardware buffer to store information about a runtime analysis of the running stages that includes their performance and measured waiting times, and a mechanism to transfer the cache state of stages that are accelerated by adapting their allocation to a faster core. As it requires shared memory, this method is not applicable to many-core systems.

The authors of [28] show that many state-of-the-art runtime resource allocation methods fail at achieving efficient allocations for scenarios that include a large number of cores for scalability reasons. Hence, the authors propose a functional performance model to estimates the performance of an application for a given problem size (input data). The focus of this work is not directly on software pipelines, but it focuses on data-intensive iterative parallel applications that follow a computational model which shares properties with software pipelines. However, this work focuses on providing a functional performance model, which is a sub-problem of resource allocation, and does not propose a concrete allocation method.

For smaller multi-core systems, [8] proposes a runtime resource allocation method for stream-processing applications that utilizes a compile-time step and a runtime dynamic resource allocator. At compile time, profiling information about the application is obtained and nodes are replicated and ordered. Their approach allows for dynamically changing the degree of parallelism of the application at runtime in order to achieve a broad applicability for different systems. However, this approach requires shared memory and is hence not applicable to many-core systems.

General Parallel Applications, Distributed Memories

Runtime resource allocation for homogeneous systems with multiple voltage levels is presented by [27]. The authors propose a two-phase method for iteratively allocating resources: in the first phase, a region of the chip that is suitable to run a newly created application is determined. In a second step, a heuristic allocation method allocates tasks to cores in this region to minimize the energy consumption that arises from inter-task communication. A major focus is put on scalability for many-core systems with a large number of cores. In contrast to the resource allocation methods proposed in this thesis, this approach assumes that there cannot be more tasks than cores in a system, which is a fundamentally different assumption which implies that resource allocation has only marginal impact on the performance of a system. Furthermore, this approach does assumes that the resource requirements of the individual tasks remain static at runtime and thus, this approach is not suitable for dynamic scenarios where this assumption is violated.

The authors of [7] present a heuristic runtime load balancing method for asynchronous, iterative algorithms (AIAC) in grid computing systems. It aims at balancing the computational load among cores by exchanging workload when imbalances are detected. To achieve this, this method repeatedly observes the workloads of all cores and performs distributed interactions among them. Workload is exchanged by transferring an application's individual work items, e.g. video frames, between neighboring cores. Due to its focus, it does not take inter-task communication into account. It therefore may achieve inferior performance when tasks communicate heavily, as it is the case for many complex, real-world applications.

In [64], DistRM, a distributed heuristic for resource allocation is presented that uses interacting software agents. Based on runtime observations and on offline profiles, agents possess local information and interact to allocate or re-allocate resources when applications are started, stopped, or when their computational requirements vary significantly. However, their approach for achieving a scalable solution for up to 4096 cores limits their decisions to local regions, which can potentially result in a low throughput of the system.

As AIAC [7] and DistRM [64] are most similar to the methods for resource allocation presented in this thesis, Section 8 compares them to the proposed methods.

General Parallel Applications, Shared Memories

A large and diverse set of methods exist for allocating resources in shared-memory systems. As an example, [21, 62, 66, 72, 75, 89, 90, 99, 110] propose adaptive resource allocation methods that aim at balancing the computational load at runtime. The authors of [108] propose to derive co-schedules of the individual tasks based on offline profiles, with an extension to support different priority levels [109]. However, the focus of these methods is on architectures with few cores and they require a shared address space and assume a small number of cores. Hence, they cannot be compared directly to the methods for resource allocation proposed in this thesis.

3.2.3 Summary of Resource Allocation Methods

To summarize, the state-of-the-art resource allocation methods that focus on design-time methods assume that all properties of a system and its running applications that are relevant to resource allocation are predictable at design time. As a consequence, it is likely that these methods cannot achieve good results in dynamic scenarios where the set of concurrently running applications, the resource requirements of the individual tasks, or the communication volume between them is unpredictable at design time. However, such scenarios are increasingly common in many instances.

To overcome these deficiencies, runtime resource allocation aims at allocating tasks based on runtime observations on the system state.

The existing methods that target dynamic scenarios, e.g. DistRM [64] and AIAC [7], do not take inter-task communication into account (AIAC [7]) or limit their decisions to local regions (DistRM [64]). Furthermore, both methods does not consider scenarios where memory-intensive applications may cause memory controllers to operate in saturation, which may have significant negative effects on the performance of a system. As this has recently shown to be a major issue in many-core systems [1], such scenarios must be taken into account. Hence, this thesis shows that significant improvements can be achieved upon these methods by jointly taking computation, communication, and the memory access behavior of applications into account and presents novel resource allocation methods to achieve this.

The other state-of-the-art methods which may be applicable to many-core systems target scenarios where applications may be started or stopped unpredictably but assume that the resource requirements of individual tasks remain static at runtime ([27, 70]). However, this is not true in many cases. Most methods for resource allocation, however, require shared memory ([8, 21, 56, 62, 66, 72, 75, 89, 90, 99, 110]) and are thus not applicable to many-core systems. Hence, the state-of-the-art methods for resource allocation do not sufficiently address dynamic scenarios in many-core systems.

3.3 Task Migration Mechanisms

Re-allocating resources at runtime may require *task migration*, i.e. the transfer of the execution of a task from a source core to a destination core at a given point of time. This usually happens after the task has already performed some computation on the source core. In the context of this thesis, task migration refers to the concrete mechanism to transfers the execution between two cores and does not include the decision process of determining the destination core or the point of time when a migration is performed.

Task migration involves transferring the *task context*, i.e. the data that characterizes the state of a task, from the source to the destination core. When source and destination cores share memory, the *task context* that must be transferred is limited to the processor registers. However, when the cores do not share memory, the task context must also include the task's heap, stack, and program code, and must be transferred from the memory that is associated with the source core to the memory that is associated with the destination core. As this thesis considers many-core systems to depart from the paradigm of sharing memory among cores globally, this section focuses on task migration for cores that do not share memory.

Task migration induces potentially significant overhead and, when involved frequently, may severely degrade the throughput of a system [4, 100, 107]. In order to minimize the overhead of task migrations, the policy *how* to transfer the task context is of crucial importance. Furthermore, if invoked frequently, task migration may contribute significantly to on-chip traffic and can thus impair the throughput of other tasks that communicate heavily. Hence, a task migration mechanism should allow a system to balance the required time versus the increased bandwidth requirements adaptively based on runtime observations.

The following discusses the state-of-the-art task migration mechanisms and discusses the particular benefits and drawbacks. Task migration may be implemented on application level and on system level.

3.3.1 Application-level Mechanisms

Task migration can be implemented through application-level checkpointing [80, 90] and application-level save-restore mechanisms [20], which is not transparent to the application. Checkpointing-based task migration allows to migrate tasks when their context can be accurately captured by the software: At predefined points of times (checkpoints), the application polls a controlling instance whether it should migrate to a destination core [2, 13]. In this case, the application is responsible to identify, collect, and transfer the relevant context required to continue the execution on the destination core. However, these approaches require an individual implementation for each application.

For applications that follow a message-based producer/consumer programming model, [90] presents a task migration mechanism where tasks are requested to empty their work queue. After this work queue is emptied, they can be restarted at a destination without migrating context information. Another approach is to enforce an application-level context save-restore mechanisms that shifts the responsibility of migrating the memory contents to the application, as proposed by [20]. This approach, however, requires a careful individual implementation for each application.

Both the message- and checkpoint-based mechanisms suffer from an unpredictable and potentially very long time between initiating and finishing a task migration. This could limit the frequency of task migrations, which is undesirable for the efficient re-allocation of resources at runtime. Furthermore, application-level mechanisms require an individual implementation which is error-prone and time consuming. Hence, system-level task migration mechanisms have been proposed which shift the responsibility of capturing and transferring the context of an application to the operating system.

3.3.2 System-level Mechanisms

System-level task migration mechanisms do not require application-level support.

In [4], a task migration mechanism for distributed operating systems is presented. It transfers the entire task context to the memory associated with the destination core. For multi-threaded tasks, all threads of a tasks are transferred at once. A system of marshalling and demarshalling of data is used and requires an *establishment* phase where pointers to memory and to kernel data structures are translated accordingly.

The mechanisms that are employed are as follows: The *eager-copy* task migration mechanism first pauses the task on the source core. Then, it sends the entire program context to the destination core and resumes the task there once this has completed. A bandwidth-reduction *lazy-copy* mechanism initially only transfers the minimum task context (program code, stack and registers) and sends missing pages when the resumed task on the destination core tries to read them. A latency-optimized *pre-copy* mechanism transfers the entire program context while the task runs on the source and re-sends pages modified after their transfer while the task is paused.

The authors of [10] propose task migration for distributed operating systems using so-called *deputy* and *remote* tasks. When a task is migrated, a *deputy* task is created at the destination core and the original task becomes a *remote* task. All system calls and memory accesses of the deputy task are then transferred (via network sockets) to the remote task. This way, the task context need not be transferred. However, as each access to the task context needs to be transferred, the overhead of this approach can be very large.

In [100], a task migration mechanism is proposed that allows task migration with significantly reduced overhead. It introduces a *post-copy* mechanism that transfers only part of the task context while the task's execution is paused, and sends other data once the task has been resumed at the destination core. The *post-copy* mechanism uses a *page fault* mechanism where, whenever a page is accessed that has not yet been transferred, the operating system *initiates* the transfer of this page.

Task migration in many-core systems, however, has fundamentally different requirements as in distributed systems. In many-core systems, task migration may be invoked frequently e.g. for achieving load balancing. Hence, the performance overhead is of crucial importance. Aspects of data consistency and security have only marginal impact as the data is not transferred over unreliable and untrustworthy networks. To achieve this, [73] proposes architectural support for task migration and requires at least one core to be idle at any time. This approach focuses on reducing the overhead of task migrations in systems with shared memory. Thus, this mechanism is not suitable for systems with distributed memory, as task migration in shared-memory systems does not require to migrate the task context between cores.

3.3.3 Summary of Task Migration Mechanisms

To summarize, the state-of-the-art task migration mechanisms do not sufficiently address the needs of many-core systems that may potentially adapt the resource allocations frequently at runtime. State-of-the-art application-level methods require an individual implementation of task migration for each application. Specifically, they require applications to identify, collect, and transfer the relevant task context. This can hinder the parallelization of applications significantly. Hence, such approaches conflict with the desire for a high degree of parallelism in many-core systems. State-of-the-art system-level task migration mechanisms, however, do not sufficiently target scenarios where applications may be migrated frequently, as the latency induced by task migration may be high in many cases.

To account for these different requirements, this thesis proposes a novel, context-aware runtime adaptive task migration mechanism, CARAT. The proposed mechanism adapts the page migration concepts from [100, 107] by incorporating a thorough memory access behavior analysis for the state-of-the-art applications. This way, a many-core system is able to adapt resource allocations at runtime.

3.4 Summary

Efficient resource allocation is key for a high performance in many-core systems. In dynamic scenarios, resource allocation must be able to calculate and adapt allocations at runtime with a minimum performance overhead.

However, the state-of-the-art resource allocation methods either require shared memory, do not sufficiently address dynamic scenarios, or do not take all important factors into account. Furthermore, the state-of-the-art mechanisms for task migration may induce a significant overhead when migrating tasks frequently.

This leaves a need for novel resource allocation methods that build on top of a task migration mechanism which is suitable for adapting resource allocations at runtime. The rest of this thesis proposes methods and mechanisms to overcome these deficiencies of the state-of-the-art.

Chapter 4

Models and Problem Definition

4.1 Application Model

The resource allocation methods that are proposed in this thesis are based on an application model. In the following, this model for software-pipelined applications is detailed.

Each application k forms a pipeline P_k with N_k stages. Every stage S_j is characterized by c_j, e_j and o_j that denote the time consumed for the computation required to process a data item, for receiving the input data from its direct predecessor, and for transferring the output data to its direct successor. Figure 4.1 illustrates this model.

In order to decide about the allocation of resources to applications, it is important to model their throughput for a given allocation. To achieve this, we require that each core belongs to at most one application (i.e. cores may not be shared among applications). Their maximum throughput is limited by their slowest stage. The *maximal response time* R_k for pipeline P_k can be defined consequently:

$$R_k = \max_{1 \leq j \leq N_k} \{e_j + c_j + o_j\}. \tag{4.1}$$

Therefore, the maximum *throughput* of pipeline P_k is defined as $\frac{1s}{R_k}$.

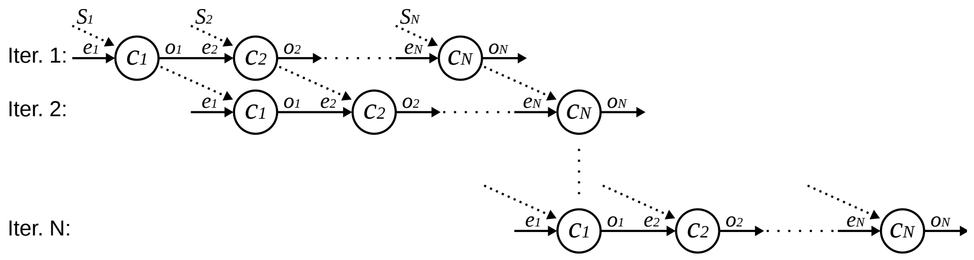


Figure 4.1: Application model for software pipelines. Each stage S_j is characterized by the time consumed by its computation c_j, by receiving its input data e_j, and by sending the output data to its direct successor o_j. The execution of the different stages of application k overlap, and different applications execute independently.

In order to change the degree of parallelism of an application at runtime, we define the basic operation *fusion* (and the inverse operation *fission*), in which multiple consecutive pipeline stages are combined, similar to fusing filters in StreamIt [115]. A fusion of stages replaces the fused stages with a new stage which combines the computational requirements of the original stages but does not require communication between them, as shown in Figure 4.2.

This way, fusing stages may reduce the maximal response time R_k of a pipeline even in cases when the total number of stages of all applications does not exceed the number of cores. Additionally, fusing stages reduces the number of stages from N_k to N'_k ($N'_k \leq N_k$), and thus it reduces the degree of parallelism of the application, which then runs on a smaller number of cores.

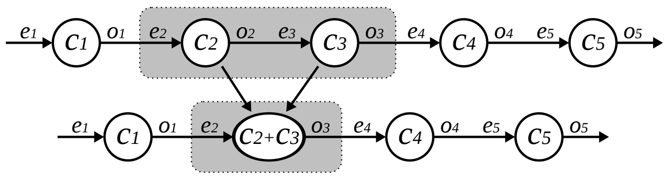


Figure 4.2: Fusion of pipeline stages. Stages can be fused at runtime. When stages are fused, they are executed on the same core and thus, no inter-core communication between fused stages is necessary.

4.1.1 Extension for Multiple Memory Islands

In order to extend this application model for systems with multiple memory islands, let us consider the following: After fusing the N_k stages of an application k, $N'_k \leq N_k$ stages remain. Per processed data item, each stage S_i^k has a *computational requirement* of $c_{k,i}$ (unit: time) and a *bandwidth requirement* of $b_{k,i}$ (unit: MB/s). This bandwidth requirement expresses the accesses of a stage to off-chip memory.

When stage S_{i-1}^k is allocated to the same memory island as S_i^k, it takes $e_{k,i}^{\text{in}}$ time for stage S_i^k to receive the data from stage S_{i-1}^k. In contrary, when stage S_{i-1}^k is allocated to a different memory island than S_i^k, it takes $e_{k,i}^{\text{out}}$ time for stage S_i^k to receive the input data from stage S_{i-1}^k. Similarly, $o_{k,i}^{\text{in}}$ represents the time to send the output data to stage S_{i+1}^k when both stages are allocated to the same memory island, and $o_{k,i}^{\text{out}}$ corresponds to the time required for sending the output data when S_i^k and S_{i+1}^k are allocated to different memory islands. Figure 4.3 illustrates the model.

Such a model is based on the assumption that $o_{k,i-1}^{\text{in}}$ and $e_{k,i}^{\text{in}}$ correspond to exchanging pointers, given that memory can be accessed in a shared manner within a memory island. Likewise, when S_{i-1}^k and S_i^k communicate across memory islands, $o_{k,i-1}^{\text{out}}$ and $e_{k,i}^{\text{out}}$ correspond to transferring the data between memory controllers, which requires considerably more time.

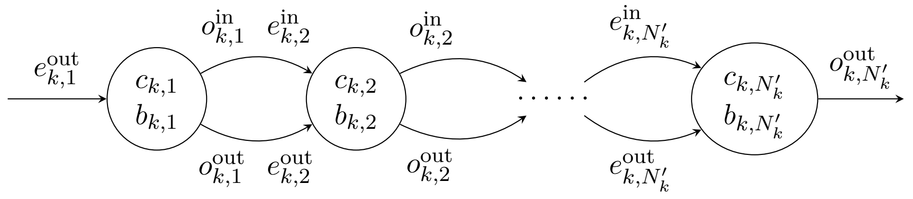


Figure 4.3: The model of software pipelines. When inter-core communication contains data that is stored in off-chip memory (which can be assumed to be true in many cases), the required time for the data transfer differs between communicating within one memory island and when communicating across memory islands. Inside a memory island, cores may share memory and can thus communicate data by exchanging pointers. When cores that belong to different memory islands communicate data, they have to transfer the data between memories.

4.2 Hardware Model

The hardware model used in this thesis is defined as follows: in a system, there are V memory controllers $\mathbf{M} = \{M_1, M_2, \dots, M_V\}$, and each controller serves the Q cores of the corresponding memory island $I_i \in \mathbf{I} = \{I_1, I_2, \dots, I_V\}$, such that there are $Q \cdot V$ cores in total. Every core $C_{i,j}$ is identified by a tuple (i, j), where $i = 1, 2, \dots, V$ represents the memory islands to which it belongs, and $j = 1, 2, \dots, Q$ the index of the core inside its memory island. Figure 4.4 shows an example of the architecture.

We consider that each memory controller M_i has a bandwidth constraint of B_i and a *remaining* bandwidth constraint of B'_i, such that when no stages are allocated then $B'_i = B_i$ holds. When allocating stage S^k_h from pipeline P_k to

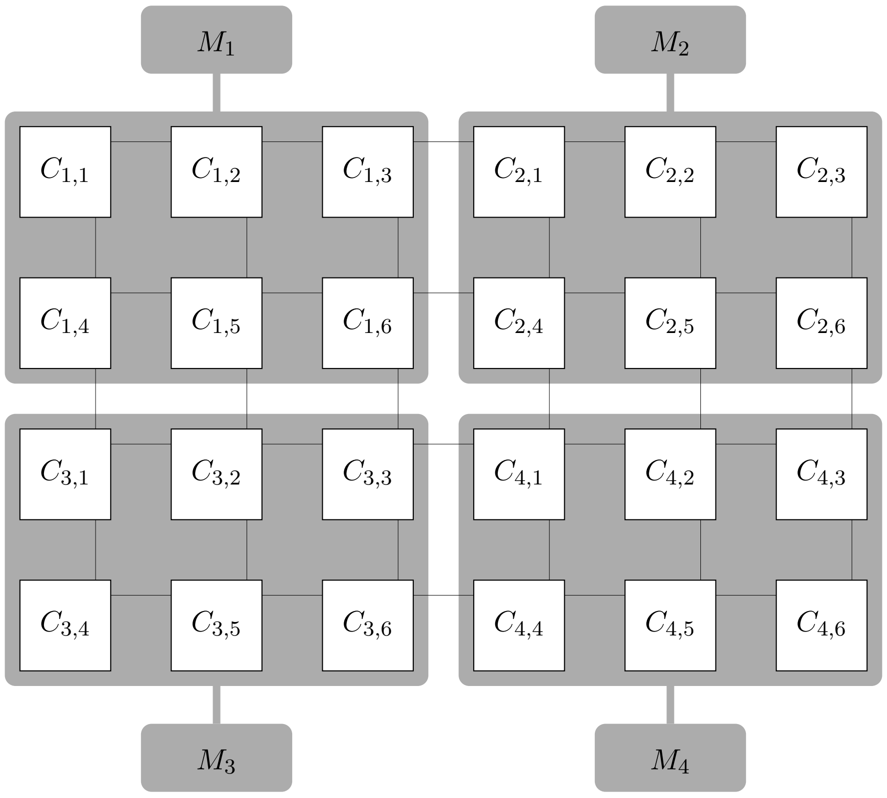


Figure 4.4: Example of a target architecture showing four memory islands. Each island I_i contains 6 cores $C_{i,j}$ and one memory controller M_i. All cores are connected via a network-on-chip.

memory island I_i, the value of B'_i is updated by subtracting the bandwidth requirement $b_{k,h}$.

Similarly to B'_i, Q'_i denotes the number of *free cores* (i.e. cores where no pipeline stage has been allocated to) of memory island I_i. When n stages of a pipeline are allocated to island I_i, then n is subtracted from Q'_i. Obviously, $Q'_i \geq 0$ holds, because only up to Q fused stages can be allocated to an island.

4.3 Problem Definition

Let us define the problem of allocating resources to software pipelines. We divide the problem into:

1. How to assign the stages of an application to a given number of cores (Section 4.3.2), thus providing the fusions of the pipeline stages (Sub-Problem).
2. How to distribute the cores of a system among the applications (Section 4.3.1) so that the overall system throughput is maximized (Global Problem).

4.3.1 Optimizing System Throughput

Given a set of K *weighted* applications $P = \{P_1, P_2, \ldots, P_K\}$ where the weights $W = \{w_1, w_2, \ldots, w_K\}$ express priority levels, each application P_k uses up to M_k cores and has a maximal response time R_k. The objective is to **maximize the overall weighted system throughput** by finding an optimal distribution of (up to) M available cores to the individual applications.

$$\text{Maximize} \left\{ \sum_{k=1}^{K} \frac{w_k}{R_k} \right\} \quad | \text{ such that } \sum_{k=1}^{K} M_k \leq M \tag{4.2}$$

In a system with Q memory islands, Equation 4.2 is formulated as

$$\text{Maximize} \left\{ \sum_{k=1}^{K} \frac{w_k}{R_k} \right\} \quad | \text{ such that } \sum_{k=1}^{K} N_k \leq Q \cdot V \tag{4.3}$$

The stages allocated to the cores of a memory island I_i may suffer from degraded performance when the corresponding memory controller operates in saturation, which means that the remaining bandwidth capacity B'_i becomes negative, i.e. $B'_i < 0$. Consequently, in addition to finding a solution for the goal and constraint of Equation (4.3), the goal is to avoid such a saturation by balancing the load among memory controllers. Finding a balanced memory bandwidth assignment is a *NP-complete* problem in a strong sense, which can be easily reduced from the 3-PARTITION problem [36].

Alternative definition: The definition of the Global Problem (Equations 4.2 and 4.3) aims at maximizing the overall weighted throughput. However, this implies that applications with low weights may suffer from very low throughputs in favor of the throughput of applications with high weights. This may be unacceptable in scenarios where a guaranteed minimum throughput for each application is required.

In order to achieve this, the Global Problem can be formulated alternatively:

$$\text{Maximize } \left\{ \min_{1 \leq k \leq K} \left\{ \frac{w_k}{R_k} \right\} \right\} \quad | \text{ such that } \sum_{k=1}^{K} M_k \leq M \tag{4.4}$$

This ensures a minimum target performance for each application.

To solve the Global Problem, we present a centralized, optimal method in Section 5.2 and a highly scalable, distributed method in Section 5.3. However, the sub-problem of fusing pipeline stages needs to be solved first:

4.3.2 Fusion of Pipeline Stages

The throughput of an application is affected by *how* the stages are fused. Thus, we define a sub-problem for the fusions of each application $k \in K$ to minimize the maximal response time R_k.

For this, for each application k, we define a set of F_k fusions:

$$\{F_1(1, j_1), F_2(j_1 + 1, j_2), \ldots, F_{F_k}(j_{F_k - 1} + 1, N_k)\} \quad | \text{ such that } F_k \leq M_k$$

so that each application k uses not more than M_k cores. Furthermore, according to Section 4.1, we define

$$\begin{aligned} F_i(l,j) = e_l + o_j + \sum_{h=l}^{j} c_h& \\ 1 \le l \le N_k,& \\ l \le j \le N_k& \end{aligned}$$

and thus, we define the sub-problem to

$$Minimize \left\{ \max_{1 \le f \le F_k} \{F_f\} \right\} \quad | \; \forall k \in K \tag{4.5}$$

We present an algorithm to solve this problem in Section 5.1.

4.4 Summary of Models and Problem Definition

To summarize, allocating resources requires to allocate the cores of a system to the applications (Equations 4.2, 4.3, and 4.4), and to fuse the stages of each application so that $F_k \le M_k$ (Equation 4.5). Chapters 5 and 6 detail how this can be achieved.

Some terms and definitions were introduced in this chapter and will be used extensively in the rest of this thesis. For a quick reference, these terms are:

1. The time required for the computations of processing a data item is denoted by c_i and is referred to as the *computational requirements* of a stage i, assuming the responsible memory controller is not saturated.

2. The *bandwidth requirements* b_i of a stage i denote the amount of off-chip memory that it accesses, in MB/s, assuming that its throughput is not limited by the bandwidth constraints of its memory controller. The *memory requirements* of a stage denote this per data item, in MB.

3. The *throughput* of a software pipeline denotes the number of data items the pipeline finishes per second. The *system throughput* is the average throughput of all applications.

4. *Fusing* consecutive stages replaces them with a new stage that combines their computational requirements, similar to fusing filters in StreamIt [115]. This reduces the degree of parallelism of the pipeline

and the number of cores it uses. No on-chip communication between fused stages is necessary. Stages can be fused (and fused stages can be split) at runtime.

5. A *memory island* consists of one memory controller and a number of cores. All memory accesses of the cores of one memory island are served by the same memory controller. The tasks that are allocated to one memory island can share memory and may pass pointers to data, while tasks allocated to different memory islands transfer data via message passing (MPI, e.g. [94]).

6. The *load* of a memory controller denotes the accesses it serves per second, in MB/s.

7. The *bandwidth constraint* of a memory controller expresses the maximum load, in MB/s.

Chapter 5

System-controlled Resource Allocation

System-controlled methods employ one (centralized methods) or more (distributed methods) controlling instances. In the following, the system-controlled methods for resource allocation CeRA (5.2), DiRA (5.3) and MOMA (5.4) are proposed that have been published in [54] and [55]. These methods build on our proposed algorithm for optimally fusing the stages of a software-pipelined application for a given number of cores (5.1).

5.1 Fusing Pipeline Stages

To find an optimal solution to the sub-problem of fusing pipeline stages, all possible combinations of fusions have to be taken into consideration. An exhaustive search would lead to an exponential time complexity, which may be unacceptable especially for adapting the allocation of resources at runtime. Hence, an algorithm based on dynamic programming is proposed in the following. The proposed algorithm derives optimal solutions for minimizing the maximal response time of each pipeline P_k using $m = M_k$ cores.

Let $P_{k,j}$ be a *sub-pipeline* of P_k that is formed by the stages S_1 to S_j of pipeline P_k. The dynamic programming is then defined as a recursive function $R_k(j, m)$ that stores the optimal configurations for minimizing the maximal response time of $P_{k,j}$ with at most $m = M_k$ cores. That is, let $R_k(j, m)$ be the minimum maximal response time for executing $P_{k,j}$ on m cores. Moreover, a table $F_k(\ell, j)$ is constructed for all ℓ, j such that $1 < \ell \leq j \leq N_k$, in which

$$F_k(\ell, j) = e_\ell + o_j + \sum_{h=\ell}^{j} c_h. \tag{5.1}$$

Then, the initial boundary conditions for $R_k(j, 0)$ and $R_k(j, 1)$ are:

$$\begin{array}{ll} R_k(j, 0) = \infty & \forall j = 1 \dots N_k \\ R_k(j, 1) = F_k(1, j) & \forall j = 1 \dots N_k \end{array} \tag{5.2}$$

Furthermore, we define function $\text{minmax}RF_k(j, m)$ as:

$$\text{minmax}RF_k(j, m) = \min_{m-1 \leq \ell < j} \{\max\{R_k(\ell, m-1), F_k(\ell+1, j)\}\}. \tag{5.3}$$

The recursive function for $R_k(j, m)$ with $m \geq 2$ is defined as:

$$R_k(j, m) = \begin{cases} R_k(j, m-1) & j < m \\ \min\{R_k(j, m-1), \text{minmax}RF_k(j, m)\} & j \geq m \end{cases} \tag{5.4}$$

The proposed algorithm starts by computing the resulting maximal response times utilizing only one core for the first $j = 1 \dots N_k$ stages. Then, the maximal response times for the first $j = 1 \dots N_k$ stages on up to two cores is computed. Since the maximal response times of using only one core for the first j stages has already been computed and is stored in F_k, it can be decided whether to use one or two cores (in one of the possible fusion combinations) for the same j stages without recomputing previous results recursively.

The process is repeated for three and up to M cores. As the table F_k contains the information whether it is optimal to use one or two cores for the first j stages, only the previous result must be compared with any new possible fusion for the same j stages on up to three cores. Thus, iteratively, an optimal solution is computed because all combinations of stages and cores are considered, but the complexity is reduced since optimal solutions are stored in tables and do not need to be recomputed.

The space/time complexity is $O(N_k^2)$ for building the table F_k. The time complexity for building an entry $R_k(j, m)$ is $O(j) = O(N_k)$. The size of the table $R_k(j, m)$ is $O(M_k N_k)$. Therefore, the total time complexity is $O(M_k N_k^2)$. The maximal response time by using at most M_k cores for

Algorithm 1: Minimizing the Maximal Response Time

```
Input: The computational requirements $e$, $c$ and $o$ for the $N_k$ stages
       of pipeline $P_k$, and the maximum $M_k$ cores available;
Result: The minimal maximal computational requirements using at
        most $M_k$ cores;

Initialize table $F_k(\ell, j)$ according to Equation (5.1),
$\forall\,(\ell, j)$ such that $1 \leq \ell \leq j \leq N_k$;
for $m = 0$ to $M_k$ do
    for $j = 1$ to $N_k$ do
        if $m \leq 1$ then
            Build $R_k(j, m)$ according to Equation (5.2);
        else
            Build $R_k(j, m)$ according to Equation (5.4);
        end
    end
end

return $R_k(N_k, M_k)$;
```

pipeline P_k is stored in $R_k(N_k, M_k)$. Algorithm 1 shows the pseudo-code for this dynamic programming.

The actual fusions that lead to the optimal result can be derived by back-tracking table F_k or by using an additional *tracking table* $TR_k(N_k, M_k)$ of size $O(M_k N_k)$. $TR_k(j, m)$ can be defined so that each cell holds the j^* value of the sub-solution that makes the programming optimal.

For the initial condition $m = 1$, $TR_k(j, m)$ is set to zero. When $j < m$, or when $j \geq m$ and $R_k(j, m-1)$ turned out to be minimal, then $TR_k(j, m) = j$. In the case where an additional core provides improvement, $TR_k(j, m)$ will be set to the index ℓ from Equation 5.3 that made this improvement possible and therefore $TR_k(j, m) \neq j$.

The fusions that give an optimal maximal response time can be derived from table $TR_k(N_k, M_k)$ as follows: starting from cell $(j, m) = (N_k, M_k)$, the table is traversed in the direction $(TR_k(j, m), m - 1)$.

If $TR_k(j, m) = j$, this means that it is not possible to allocate more cores to the pipeline since no finer granularity can be achieved or that no additional core may improve the throughput and the sub-solution that uses one less core was already optimal.

If $TR_k(j,m) \neq j$, an additional core provides improvement, so if $TR_k(j,m)+1 = j$ then stage S_j is allocated to one core and if $TR_k(j,m)+1 < j$ all stages between $TR_k(j,m)+1$ and j (both inclusive) should be fused.

Example Given the pipeline k which is illustrated in Figure 5.1 with $N_k = 4$ stages and having available up to $M_k = 4$ cores, table $F_k(l,j)$ is built according to Equation (5.1), as stated in Algorithm 1:

$$\begin{array}{lll}
F_k(1,1) = 60 & F_k(2,2) = 110 & F_k(3,3) = 110 \\
F_k(1,2) = 150 & F_k(2,3) = 60 & F_k(3,4) = 140 \\
F_k(1,3) = 100 & F_k(2,4) = 90 & \\
F_k(1,4) = 130 & & F_k(4,4) = 70
\end{array}$$

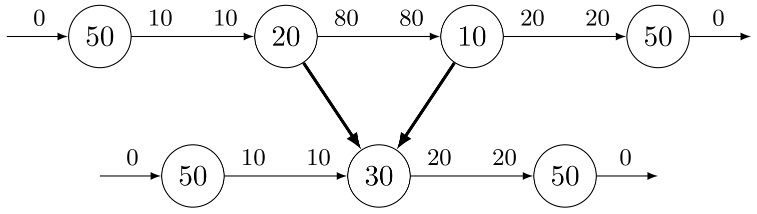


Figure 5.1: Example of fusing pipeline stages: The computational requirements of the fused stages are summed. No more inter-core communication between stages is necessary as both stages are allocated to the same core.

As a next step, the initial conditions for $R_k(j,m)$ are computed according to Equation (5.2). This means to compute the response time for a sub-pipeline with j stages using up to m cores. Since these are initial conditions, the tracking table $TR_k(j,m)$ for $m = 0,1$ has no previous value for j^*, and is therefore filled with zeros.

From now on, since $m \geq 2$, the table $R_k(j,m)$ is built according to Equation (5.4). In this particular example, when $m = 2$ the solution for every sub-pipeline chooses to use the result from $R_k(1,1)$ and to fuse the rest of the stages, thus, the tracking table $TR_k(j,2)$ will be filled with $j^* = 1$ for any j.

The results are shown in Table 5.1. The optimal solution can be derived from table $TR_k(j,m)$ by starting from cell $(j,m) = (4,4)$ and traversing the

$R_k(4,4)$: Response time

m \ j	**1**	**2**	**3**	**4**
1	60	150	100	130
2	60	110	60	90
3	60	110	60	70
4	60	110	60	70

$TR_k(4,4)$: Tracking

m \ j	**1**	**2**	**3**	**4**
1	0	0	0	0
2	1	1	1	1
3	1	2	3	3
4	1	2	3	4

Table 5.1: Example R_k and TR_k tables as used in Algorithm 1. R_k stores the minimal achievable response time for all possible fusions using $m = 1 \dots N_k$ cores considering the first $j = 1 \dots N_k$ stages. Table TR_k allows to track the optimal fusions that lead to the corresponding result in R_k.

table in the direction $(j^*, m-1)$: the optimal solution will fuse stages S_2 and S_3, and leave stages S_1 and S_4 as they are.

5.2 CeRA: Centralized Resource Allocation

With the algorithm for fusing pipeline stages of Section 5.1, the overall weighted system throughput can be maximized in a centralized manner. Suppose that $R_k(N_k, m)$ for $m = 1, 2, \dots, \min\{N_k, M\}$ has been built. For notational brevity, if $N_k < M$, we define $R_k(N_k, m) = R_k(N_k, N_k)$ for any $m \geq N_k$. Let $G(k, m)$ be the minimal weighted system response time (and consequently, $\frac{1}{G(k,m)}$ denotes the maximal weighted system throughput) for the first k pipelines based on any arbitrary order of pipelines on at most m cores. Moreover, when there is no feasible solution, i.e. $k > m$, $G(k, m)$ is defined as $-\infty$. Then, the initial (boundary) condition for $G(1, m)$ is:

$$G(1, m) = \frac{w_1}{R_1(N_1, m)} \qquad \forall m = 1, 2, \dots, M \tag{5.5}$$

The recursive function for $G(k, m)$ with $k \geq 2$ is expressed in Equation (5.6). The time complexity, provided that $R_k(N_k, m)$ is known, is $O(KM^2)$. Note that the last column of R_k, i.e. $R_k(N_k, m)$ $\forall m = 1, 2, \dots, M$, contains the

Algorithm 2: Maximizing Overall Weighted System Throughput

```
Input:  The maximum number of available $M$ cores. For every
        pipeline $P_k$, the weights $w_k$ and tables $R_k(N_k, m)$ for
        $m = 1, 2, \dots, M$;
Result: Maximum overall weighted system throughput for $K$
        pipelines, using at most $M$ cores;

for $k = 1$ to $K$ do
    for $m = 1$ to $M$ do
        if $k = 1$ then
            Build $G(k, m)$ according to Equation (5.5);
        else
            Build $G(k, m)$ according to Equation (5.6);
        end
    end
end

return $1/G(K, M)$;
```

application's weighted throughput when m cores are available. Algorithm 2 shows a pseudo-code for this dynamic programming.

$$G(k,m) = \begin{cases} -\infty & k>m \\ \max\limits_{k-1 \leq m' < m} \left\{ G(k-1, m') + \frac{w_k}{R_k(N_k, m-m')} \right\} & k \leq m \end{cases} \tag{5.6}$$

An additional tracking table $TG(K, M)$ of size $O(KM)$ allows for deriving how many cores should be allocated to each pipeline. When building $TG(k, m)$, each cell holds the m^* value of the sub-solution that makes the solution optimal. For the initial condition $k = 1$, $TG(k, m)$ is set to zero. When $k > m$, then $TG(k, m) = -1$. In the case were $k \leq m$, $TG(k, m)$ will be set to the value of m' from Equation 5.6 that made this sub-solution optimal.

Once table $TG(K, M)$ has been built, the number of cores for each pipeline can be derived from it: Starting from the final cell $(k, m) = (K, M)$, the table is traversed in the direction $(k - 1, TG(k, m))$. If $TG(k, m) = -1$, then there is no feasible solution for this set of values. In any other case, cores between $TG\,(k, m) + 1$ and m (both inclusive) should be allocated to application k.

$R_1(3, 6)$

m	$R_1(3, m)$
1	130
2	90
3	70
4	70
5	70
6	70

$R_2(5, 6)$

m	$R_2(5, m)$
1	120
2	110
3	100
4	90
5	80
6	80

$R_3(4, 6)$

m	$R_3(4, m)$
1	300
2	300
3	80
4	40
5	40
6	40

Table 5.2: Example of R_k tables of three different pipelines $P_{1\ldots3}$ for up to 6 cores. P_1 comprises 3 stages, P_2 comprises 5 stages, and P_3 comprises 4 stages. This example also illustrates that allocating $m > N_k$ cores to P_k will result in the same response time (and hence, in the same throughput) as allocating N_k cores.

5.2.1 Example

Given the pipelines R_1, R_2 and R_3 shown in Table 5.2, with weights $w_1 = w_2 = w_3 = 10000$ and having up to $M = 6$ available in the system, in order to find the maximal overall system throughput requires to compute the initial conditions for $G(k, m)$ according to Equation (5.5). Since this are initial conditions, the tracking table $TG(k, m)$ for $k = 1$ has no previous value for m^*, and is therefore filled with zeros.

From now on, since $k \geq 2$, $G(k, m)$ is built according to Equation (5.6).

The results are shown in Table 5.3. Looking at table $TG(k, m)$, starting from cell $(k, m) = (3, 6)$ and traversing the table in the direction $(k-1, m^*)$,

$G(3,6)$: Weighted System Response Time

$m \backslash k$	**1**	**2**	**3**
1	76.92	$-\infty$	$-\infty$
2	111.11	160.26	$-\infty$
3	142.86	194.44	193.59
4	142.86	226.19	227.78
5	142.86	233.76	285.26
6	142.86	242.86	410.25

$TG(3,6)$: Tracking

$m \backslash k$	**1**	**2**	**3**
1	0	-1	-1
2	0	1	-1
3	0	2	2
4	0	3	3
5	0	3	2
6	0	3	2

Table 5.3: Example G and TG tables from Algorithm 2. G stores the minimal achievable overall weighted response time (i.e. the weighted average of the response time of all applications). Table TG contains the tracking information that allows to derive how to allocate the cores to the applications so that this response time can be achieved.

one can derive that the optimal solution will allocate one core to pipeline R_1, one core to pipeline R_2 and four cores to pipeline R_3.

5.3 DiRA: Distributed Resource Allocation

The CeRA method presented in Section 5.2 is designed in a centralized manner. This requires global system knowledge and allocates the resources to all running applications en bloc. This leads to a quadratic time complexity (with the number of cores), which may be infeasible for large many-core systems. To achieve a highly scalable solution (i.e. its overhead should not grow significantly with a growing number of cores or applications), this section proposes a distributed, hierarchical method for which the pipelines are grouped into several independent clusters. Clusters are grouped hierarchically into larger clusters and so on, therefore constructing a tree, as illustrated by Figure 5.2.

There are K_0 pipelines $P_1, P_2, \ldots, P_{K_0}$ on level 0, and they form nodes are the leaves of the tree. The rest of the nodes are clusters and are expressed

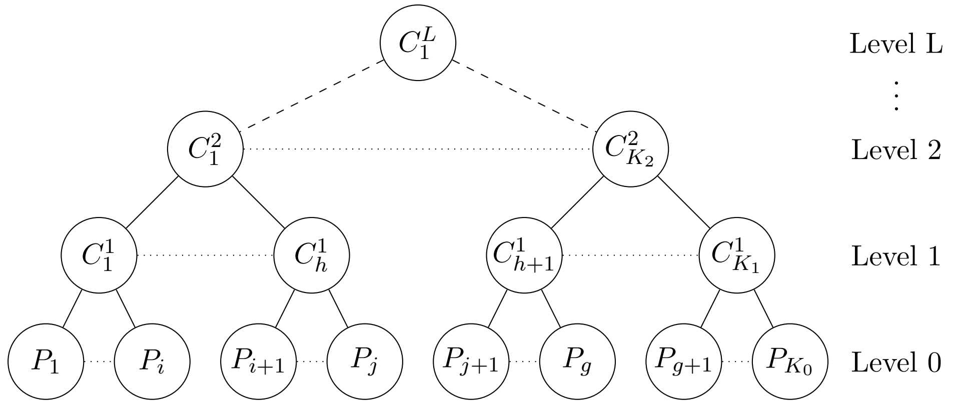


Figure 5.2: Hierarchy of our distributed DiRA resource allocation method. The applications $P_1 \dots P_{K_0}$ form the leaves of the hierarchical tree structure and are clustered on levels $1 \dots L$. The depth of the tree depends on the maximum number of children of each cluster (a design-time parameter).

as C_i^ℓ, where indexes ℓ and i represent the level of the cluster in the tree and the index of the cluster in the level, respectively. All clusters in level 1 ($\ell = 1$) are the adjacent parents of the pipelines. There are L levels in the tree, where level L is the root of the tree, and each level ℓ holds K_ℓ nodes.

With this distributed model, the solution from Section 5.1 is utilized to build the tables $R_k(N_k, M)$ for every pipeline P_k, where M continues to be the total amount of cores available in the system.

Each cluster C_i^1 (level 1) contains the information of the weights w_k and N_k columns of tables R_k, namely $R_k(N_k, M)$ of its children (pipelines) and utilizes the solutions of Sections 5.2 to build the corresponding tables $G(K^*, M)$, where K^* is the number of child nodes of the cluster. This table contains the best configuration for cluster C_i^1 by allocating $m = 1, 2, \dots, M$ cores to its children pipelines, independently of the other clusters of the same level.

Similarly, the clusters C_i^2 (level 2) contain the information of table $G(K^*, M)$ of its child clusters C_i^1 (level 1). This applies likewise to all upper levels. In this way, each level allocates cores among its children based solely on this (limited) information. Consequently, the computational requirement is distributed hierarchically among the system.

$G_i^\ell(k, m)$ denotes the table for the modified version of the dynamic programming in Section 5.2. Considering that w_1^*, R_1^*, N_1^* are the parameters of the first child (pipeline) of node C_i^1, node $C_{1^*}^{\ell-1}$ is the first child of node C_i^ℓ, value $K_{\ell-1}^*$ is the number of children of node C_i^ℓ, and value $K_{\ell-2}^*$ is the number of children of node $C_{1^*}^{\ell-1}$ and node $C_k^{\ell-1}$, then the initial conditions of $G_i^\ell(1,m)$ are:

$$\begin{aligned} G_i^\ell(1,m) &= \tfrac{w_1^*}{R_1^*(N_1^*,m)} && \forall m = 1,2,\ldots,M \quad \text{when } \ell = 1 \\ G_i^\ell(1,m) &= G_{1^*}^{\ell-1}(K_{\ell-2}^*,m) && \forall m = 1,2,\ldots,M \quad \text{when } \ell \geq 2, \end{aligned} \tag{5.7}$$

the value of $G_i^\ell(k,m)$ is set to $-\infty$ whenever $k > m$, the recursive function when $\ell = 1$ and $k \leq m$ is:

$$G_i^\ell(k,m) = \max_{k-1 \leq m' < m} \left\{ G_i^\ell(k-1,m') + \tfrac{w_k}{R_k(N_k,m-m')} \right\}, \tag{5.8}$$

the recursive function when $\ell \geq 2$ and $k \leq m$ is:

$$G_i^\ell(k,m) = \max_{k-1 \leq m' < m} \left\{ G_i^\ell(k-1,m') + G_k^{\ell-1}\left(K_{\ell-2}^*, m-m'\right) \right\}, \tag{5.9}$$

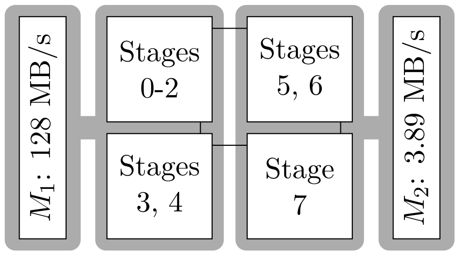
and finally, the result is found in cell $G_i^\ell\left(K_{\ell-1}^*, M\right)$.

It is important to note that even though the root node makes decisions that affect every pipeline, this is still a distributed and scalable method, since every node only contains the partial information of its children.

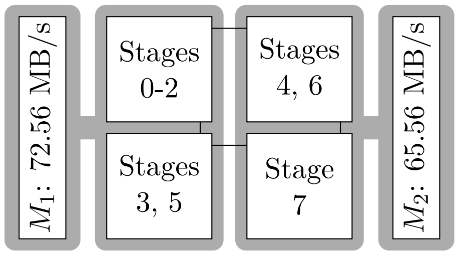
5.4 MOMA: Allocation for Multiple Memory Islands

The proposed centralized resource allocation method CeRA (Section 5.2) and its distributed, hierarchical extension DiRA (Section 5.3) as well as many state-of-the-art methods for resource allocation aim at balancing the computational load among cores jointly with inter-task communication (see [21, 62, 70]). Due to the rapidly increasing number of cores, the bandwidth limitations of *memory controllers* (i.e. controllers located on-chip that enable access to off-chip memory) may have a significant impact on the system throughput when running memory-intensive tasks [1].

This limitation results mainly from the limited number of pins to connect the chip with off-chip memory as well as from the constrained bandwidth of each



(a) Throughput: 10.71 $^1/_s$



(b) Throughput: 19.1 $^1/_s$

Figure 5.3: A system with two memory islands and four cores. For this example, the bandwidth constraint of each memory controller is 128 MB/s. Allocating resources to balance the computational requirements (a) forces the memory controller M_1 to operate in saturation, which results in a significantly reduced throughput (bound by memory bandwidth) as compared to (b) (bound by computation).

individual pin [61]. As a consequence, the number of memory controllers and their individual bandwidth are also limited. In systems with a large number of cores, each memory controller has to serve (too) many requests [1]. As an example, Intel's newest Xeon Phi™5110P integrates 16 memory controllers for 61 cores and there, each memory controller serves the accesses of approx. 4 cores [31]. However, as the number of memory controllers is limited by the pin count constraints [1], each memory controller may need to serve the accesses of 64 cores in a system with 1024 cores. Thus, it has to serve 16 times the requests.

In case one memory controller serves many memory-intensive tasks, it may operate in *saturation*, i.e. it may be requested to access more data than it can provide. This reduces the bandwidth that is available for the individual tasks, and hence their throughput will degrade. Such a saturation of memory controllers has recently been identified as a major cause for deteriorated throughput [61]. Furthermore, when tasks communicate that are allocated to cores which belong to different memory islands, the corresponding data have to be copied from one memory controller to the other. Such data transfer can be significant and may lead to severe performance penalties.

As an example of resource allocation (for the purpose of balancing computations among cores) that puts the memory controllers in saturated operation and causes a severe throughput degradation, let us consider a system with two memory islands of 2 cores each, and each memory controller with bandwidth constraint of 128 MB/s[1]. We allocate a software-pipelined application ("Object Tracking") with 8 stages. Table 5.4 lists their *computational requirements* and *memory requirements*.

Stage	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Computational requirements [ms]	24.3	11.7	13.2	22.9	26.2	23.8	26.9	49.3
Memory requirements [MB]	0.18	0.23	0.46	5.43	5.65	0.17	0.17	0.02

Table 5.4: Computational demands and memory requirements of the 8 stages of a software-pipelined "Object Tracking" application.

To balance the *computational requirements*, one may allocate Stages 0 to 2 to Core 0, stages 3-4 to Core 1, stages 5-6 to Core 2, and stage 7 to Core 3. If the memory controllers of both memory islands were not saturated, a throughput of 20.04 $^1/_s$ could be achieved. However, the resource allocation results in a load of 11.95 $^{MB}/_{dataitem}$ and 0.36 $^{MB}/_{dataitem}$ for the memory controllers, respectively. Due to their bandwidth constraint of 128 MB/s, one memory controller is saturated at approx. 10.71 $^1/_s$, which limits the application's throughput to this level. In order to account for the bandwidth constraint of memory controllers, Stages 0 to 2 may be allocated to Core 0, 3 and 5 to Core 1, 4 and 6 to Core 2, and stage 7 to Core 3, which results in a total memory requirement of 6.78 $^{MB}/_{dataitem}$ and 6.12 $^{MB}/_{dataitem}$ for the memory controllers, respectively. This resource allocation results in a throughput of approx. 19.1 $^1/_s$ (limited by the computational load on Core 2), which causes a load of 72.65 $^{MB}/_s$ and 65.56 $^{MB}/_s$ for the memory controllers, which is below their bandwidth constraint and thus, they are not operating in saturation. This corresponds to an increased throughput by approx. 76%. Figure 5.3 shows these resource allocations and the resulting throughputs.

Consequently, it is crucial that resource allocation *jointly* balances the computational requirements and the load of memory controllers. Otherwise, a saturation of memory controllers and a significantly reduced throughput can be the result. This problem worsens with a growing number of cores per memory island.

However, it is challenging to allocate resources to memory-intensive tasks

[1]We chose a low bandwidth constraint for this example to illustrate the problem in a simplified way with four cores. However, the problem arises equally in systems with fast memory controllers where each memory controller has to serve a multitude of cores.

Algorithm 3: MOMA Algorithm

```
Input: The information of the system and the pipelines to allocate;
Result: A allocation to maximize the overall system performance;

Execute Phase 1;
repeat
    Execute Phase 2;
    Execute Phase 3;
until All stages of all pipelines are allocated to cores;

return The allocation of all stages from all pipelines;
```

jointly based on computation, communication, and off-chip memory accesses because the throughput depends on the saturation of memory controllers, and vice versa.

To address this issue, MOMA is proposed as an extension to CeRA that allocates resources for memory-intensive software-pipelined applications.

5.4.1 Load Balancing of Memory Controllers

This section describes our three-phase heuristic, MOMA, presented in Algorithm 3, that finds a solution for Equation (4.3), i.e., it solves the defined problem. As an overview of MOMA, an initial solution is derived (Phase 1), and all stages have been allocated with the objective of minimizing a saturation of memory controllers (Phase 2), and minimizing the degradation of throughput that results from allocating communicating stages to multiple memory islands while not saturating the controllers (Phase 3).

5.4.2 Obtaining an Initial Solution (Phase 1)

In this phase, an initial solution based on CeRA (Section 5.2) is obtained, which solves two problems: (a) it allocates the cores to the applications, and then, given the number of cores for each application, (b) it fuses their stages so that their throughput is maximized. The initial solution neglects the bandwidth constraint of the memory controllers and the overhead when stages communicate that are allocated to different memory islands. It should be noted that our approach is not tied to this algorithm; other algorithms to obtain these initial fusions could be used as well.

The *objective* of this phase is to consider the fusion of the stages individually for all pipelines, which will not be modified from this point on. Thus, for the rest of this section, each pipeline P_k will consist of N'_k fused stages, and the constraint $\sum_{k=1}^{K} N'_k \leq Q \cdot V$ is satisfied. Given that the optimal solution that considers all constraints might result in different fusions and cores per application, no further modifications to these fusions implies that no optimal solution is achievable, but the complexity of the problem is considerably reduced. Phase 1 thus gives a starting point for phases 2 and 3, namely the fusions and number of cores for each application. Then, as explained in Section 4.1.1, the upper bound for the bandwidth requirements $b_{k,i}$ for all stages can be computed based on the memory requirements for each stage and the *maximal response time* of each pipeline.

Before proceeding to phase 2, three ordered tables are built to be used in phases 2 and 3.

Memory Controller Information (MCI) table Each row in this table holds the memory island identifier I_i, the number of free cores Q'_i of each memory island I_i and the remaining memory bandwidth constraint B'_i of the corresponding memory controller M_i, for all $i = 1, 2, \ldots, V$. The table is ordered in a decreasing manner with respect to B'_i. As explained in Section 4.2, the values of B'_i and Q'_i are updated when stages are allocated to I_i, after which the corresponding row is reordered (the entire table does not need to be re-sorted, because only this row changes). When memory island I_i has no more free cores, i.e. $Q'_i = 0$, the memory controller M_i is removed from the table. An example for this table is presented in Table 5.5.

Remaining Bandwidth (RB) requirement per pipeline table) Each row of RB contains the summed bandwidth requirements of the non-allocated stages of each pipeline P_k, for all $k = 1, 2, \ldots, K$. The table is sorted with respect to the summed bandwidth requirements in a decreasing order. When a stage is allocated to a core, its bandwidth requirements are subtracted from the corresponding row in the table, and this row is reordered (the table does not need to be re-sorted, because only this row changes). When all stages of pipeline P_k have been allocated, the corresponding row is removed from the table. An example for this table is presented in Table 5.6.

I	$\mathbf{B'}$	$\mathbf{Q'}$
I_1	B'_1	Q'_1
I_2	B'_2	Q'_2
I_3	B'_3	Q'_3
$\vdots$	$\vdots$	$\vdots$
I_V	B'_V	Q'_V

Table 5.5: Example of Table *MCI* (with $B'_1 \geq B'_2 \geq \cdots \geq B'_V$)

Single Stage Bandwidth Requirements (SSB) table This table contains the memory bandwidth requirements of every non-allocated stage of each pipeline P_k, for all $k = 1, 2, \ldots, K$. The table is ordered in a decreasing order with respect to the bandwidth requirement of each stage. When a stage is allocated to a core, the corresponding row is removed from the table. No reordering is necessary. An example for this table is presented in Table 5.7.

Phases 2 and 3 mostly focus on the first rows of each table, hence, for simplicity in presentation, the *parameters that represent the first row of each table* are denoted as $M^{\text{first}}_{\text{MCI}}$, $B'^{\text{first}}_{\text{MCI}}$, $Q'^{\text{first}}_{\text{MCI}}$, $P^{\text{first}}_{\text{RB}}$, $\sum b^{\text{first}}_{\text{RB}}$, $P^{\text{first}}_{\text{SSB}}$, $S^{\text{first}}_{\text{SSB}}$ and $b^{\text{first}}_{\text{SSB}}$.

5.4.3 Limiting Bandwidth Excess (Phase 2)

For the fusions of *Phase 1*, there may be stages whose bandwidth requirements exceed the maximum remaining bandwidth constraint among all memory controllers, i.e., $\exists b_{k,h} > B'_i$ for all $i = 1, 2, \ldots, V$, $k = 1, 2, \ldots, K$ and $h = 1, 2, \ldots, N'_k$. This means that allocating such stages to *any* memory island will saturate its memory controller as there will be at least one controller with $B'_i < 0$.

The *objective* of this phase is to balance the saturation caused by such stages

P	$\sum \mathbf{b}$
P_1	$\sum_{i=1}^{N_1} b_{1,i}$
P_2	$\sum_{i=1}^{N_2} b_{2,i}$
P_3	$\sum_{i=1}^{N_3} b_{3,i}$
$\vdots$	$\vdots$
P_K	$\sum_{i=1}^{N'_k} b_{K,i}$

Table 5.6: Example of Table *RB* (before Phase 2, and $\sum_{i=1}^{N_1} b_{1,i} \geq \sum_{i=1}^{N_2} b_{2,i} \geq \cdots \geq \sum_{i=1}^{N'_k} b_{K,i}$)

P	**S**	**b**
P_1	S_1^1	$b_{1,1}$
P_1	S_2^1	$b_{1,2}$
P_2	S_1^2	$b_{2,1}$
$\vdots$	$\vdots$	$\vdots$
P_K	$S_{N'_k}^K$	b_{K,N'_k}

Table 5.7: Example of Table *SSB* (with $N_1 = 2$ and $b_{1,1} \geq b_{1,2} \geq b_{2,1} \geq \cdots \geq b_{K,N'_k}$).

Algorithm 4: MOMA: Phase 2

```
Input: Tables $MCI$, $RB$ and $SSB$;
Result: Allocation of current high bandwidth requirement stages;

while $b_{SSB}^{first} > B'^{first}_{MCI}$ do
    Allocate single stage $S_{\mathrm{SSB}}^{\mathrm{first}}$ to the island of controller $M_{\mathrm{MCI}}^{\mathrm{first}}$;
    Update and re-order table MCI;
    Update and re-order table RB;
    Remove first row of table SSB;
    Update parameters that represent the first row of each table;
end

return Information about the allocated stages;
```

among all memory controllers by using a *"largest bandwidth requirements first"* strategy in order to minimize the impact of the saturation of memory controllers. Otherwise, $-B'_i$ may be unnecessarily high, which could lead to a severely degraded throughput for all stages allocated to the corresponding memory island I_i.

Therefore, this phase starts by checking if the stage with the highest bandwidth requirement exceeds the remaining bandwidth of the memory controller with the highest remaining bandwidth, i.e., whether the value of $b_{\mathrm{SSB}}^{\mathrm{first}}$ is larger than the value of $B'^{\mathrm{first}}_{\mathrm{MCI}}$ (given that both tables are ordered, it is not necessary to check for all i, k and h). If this condition holds, this stage is allocated to the corresponding memory island, i.e., stage $S_{\mathrm{SSB}}^{\mathrm{first}}$ is allocated to $I_{\mathrm{MCI}}^{\mathrm{first}}$.

Once the stage is allocated, the value of $b_{\mathrm{SSB}}^{\mathrm{first}}$ is subtracted from $B'^{\mathrm{first}}_{\mathrm{MCI}}$ and $Q'^{\mathrm{first}}_{\mathrm{MCI}}$ is decreased by 1. Next, the first row of table *MCI* is reordered. Similarly, $b_{\mathrm{SSB}}^{\mathrm{first}}$ is subtracted from $\sum b_{\mathrm{RB}}^{\mathrm{first}}$ and the first row of table *RB* is reordered. Finally, the first row of table *SSB* is removed, and all parameters that represent the first row of each table are updated.

If there are no more stages in table *SSB*, the allocation is completed. If there are still stages in table *SSB*, the process is repeated until $b_{\mathrm{SSB}}^{\mathrm{first}}$ is smaller than $B'^{\mathrm{first}}_{\mathrm{MCI}}$, and then proceed to Phase 3. A pseudo-code for Phase 2 is presented in Algorithm 4.

5.4.4 Allocation for Maximizing Througput (Phase 3)

The objective of this phase is to allocate consecutive stages of a pipeline to a memory island, such that the *used bandwidth* of its memory controllers is maximized without exceeding the bandwidth constraint. Stages are allocated considering both bandwidth requirements and the communication latency when communicating between memory islands.

This phase focuses on the pipeline with the highest bandwidth requirement for non-allocated stages and the highest remaining bandwidth constraint, i.e., pipeline $P_{\mathrm{RB}}^{\mathrm{first}}$ and controller $M_{\mathrm{MCI}}^{\mathrm{first}}$, respectively. For simplicity in presentation, we consider $P_j = P_{\mathrm{RB}}^{\mathrm{first}}$ and denote $N_j^{\text{non-allocated}}$ as the number of non-allocated (fused) stages of pipeline P_j.

Next, all combinations of allocating $1, 2, \ldots, \min\left\{N_j^{\text{non-allocated}}, Q'^{\mathrm{first}}_{\mathrm{MCI}}\right\}$ consecutive stages of pipeline P_j to the memory island of controller $M_{\mathrm{MCI}}^{\mathrm{first}}$ are evaluated. After Phase 2, at least one combination that requires less bandwidth than $B'^{\mathrm{first}}_{\mathrm{MCI}}$ exists, e.g., any single stage $b_{j,h}$ for a $1 \leq j \leq K$ and $h = 1, 2, \ldots, N_j^{\text{non-allocated}}$. Furthermore, only the combinations of consecutive stages of pipeline P_j are considered that have less or equal bandwidth requirements than $B'^{\mathrm{first}}_{\mathrm{MCI}}$. Therefore, when checking the combinations of allocating stages S_h^j to S_ℓ^j, if $\sum_{n=h}^{\ell} b_{j,n} > B'^{\mathrm{first}}_{\mathrm{MCI}}$, there is no need in considering the rest of the combinations that start on S_h^j.

In order to consider B'_i as well as $e_{j,s}^{out}$ and $o_{j,s}^{out}$ when allocating stages to memory controllers, a window that contains different possible combinations with similar bandwidth requirements is introduced. The combination that results in the highest throughput for its application is chosen (affected by $e_{j,s}^{out}$ and $o_{j,s}^{out}$). For all possible combinations in the window $\left[B'^{\mathrm{first}}_{\mathrm{MCI}} - \frac{B'^{\mathrm{first}}_{\mathrm{MCI}}}{Y}, B'^{\mathrm{first}}_{\mathrm{MCI}}\right]$, where Y is any integer (a design parameter) larger or equal than 1 that sets the size of the window. In other words, e.g., with $Y = 5$, only the combinations that require a bandwidth between 80% and 100% of $B'^{\mathrm{first}}_{\mathrm{MCI}}$ are considered. If there is no combination inside this window, the window is moved to $\left[B'^{\mathrm{first}}_{\mathrm{MCI}} - 2\frac{B'^{\mathrm{first}}_{\mathrm{MCI}}}{Y}, B'^{\mathrm{first}}_{\mathrm{MCI}} - \frac{B'^{\mathrm{first}}_{\mathrm{MCI}}}{Y}\right]$, e.g., 60% and 80% of $B'^{\mathrm{first}}_{\mathrm{MCI}}$ with $Y = 5$. This is repeated up to Y times until at least one combination inside the window is found. Among all combinations inside the evaluated window, the one with the minimum *partial maximum response time* for consecutive stages S_h^j to S_ℓ^j is chosen according to Equation (5.10). Then, these stages are allocated to the memory island of controller $M_{\mathrm{MCI}}^{\mathrm{first}}$.

$$T'_k(h,\ell) = \begin{cases} e^{\text{out}}_{k,h} + c_{k,h} + o^{\text{out}}_{k,h} & \text{if } h = \ell \\ \max\left\{ \begin{array}{c} e^{\text{out}}_{k,h} + c_{k,h} + o^{\text{in}}_{k,h}, \\ \max\limits_{h<n<\ell} \left\{ e^{\text{in}}_{k,n} + c_{k,n} + o^{\text{in}}_{k,n} \right\}, \\ e^{\text{in}}_{k,\ell} + c_{k,\ell} + o^{\text{out}}_{k,\ell} \end{array} \right\} & \text{if } h < \ell \end{cases} \tag{5.10}$$

Once stages S^k_h to S^k_ℓ are allocated, table *MCI* is updated by subtracting $\sum_{n=h}^{\ell} b_{k,n}$ from $B'^{\text{first}}_{\text{MCI}}$ and reorder the first row. For updating tables *RB* and *SSB*, some further considerations need to be taken. In case that $h = 1$ or $k = N^{\text{non-allocated}}_k$, Phase 3 proceeds in a similar fashion as done for Phase 2: $\sum_{n=h}^{\ell} b_{k,n}$ is subtracted from $\sum b^{\text{first}}_{\text{RB}}$, and the first row of the table *RB* is reordered; and all the rows that correspond to the allocated stages from table *SSB* are removed.

However, when $h > 1$ and $k < N^{\text{non-allocated}}_k$, it implies that some stages are allocated to the memory island of controller $M^{\text{first}}_{\text{MCI}}$, while both their preceding stages as well as their succeeding stages are allocated to different islands. When this happens, in order to correctly compute the minimum *partial maximum response time* for Phase 3, pipeline P_k needs to be split into two sub-pipelines, one with stages S^k_1 to S^k_{h-1} and another with stages $S^k_{\ell+1}$ to $S^k_{N'_k}$. Failing to do this would give incorrect results for Equation (5.10) for Phase 3. With the sub-pipelines, table *RB* is updated by removing the old pipeline and inserting the new sub-pipelines in the corresponding order. Similarly, all stages from the previous pipeline are removed from table *SSB* and the stages of the sub-pipelines are inserted, also in the corresponding order.

Once all three tables are updated, all parameters that represent the first row of each table are updated as well. If there are no more stages in table *SSB*, the allocation is completed. If there are still stages in table *SSB*, the algorithm continues to execute by returning to Phase 2, since after updating $B'^{\text{first}}_{\text{MCI}}$, now there may exist a single stage with a $b^{\text{first}}_{\text{SSB}}$ value larger than $B'^{\text{first}}_{\text{MCI}}$. A pseudo-code for Phase 3 is presented in Algorithm 5.

5.4.5 Algorithmic Complexity

For notational simplicity, $N_{\max} = \max\limits_{1\le j\le K} N_j$. The time complexity for Phase 1 is for computing the initial solution using the CeRA method described in

Algorithm 5: MOMA: Phase 3

```
Input: Tables $MCI$, $RB$ and $SSB$;
Result: Allocation of stages for highest bandwidth pipeline;

$P_k \leftarrow P_{\mathrm{RB}}^{\mathrm{first}}$;

for $h = 1, 2, \ldots, \max\left\{N_k^{non\text{-}allocated}, Q'^{first}_{MCI}\right\}$ do
    for $\ell = h, h+1, \ldots, \max\left\{N_k^{non\text{-}allocated}, Q'^{first}_{MCI}\right\}$ do
        if $\sum_{n=h}^{\ell} b_{k,n} \leq B'^{first}_{MCI}$ then
            for $y = 1, 2, \ldots Y$ do
                if $1 - \frac{y}{Y} \leq \frac{\sum_{n=h}^{\ell} b_{k,n}}{B'^{first}_{MCI}} \leq 1 - \frac{y-1}{Y}$ then
                    Window[$y$] $\leftarrow$ Append combination $\left[S_h^k, S_\ell^k\right]$;
                end
            end
        end
    end
end

for  each $y = 1, 2, \ldots Y$ do
    if Window[$y$] is not empty then
        for each element in Window[$y$] do
            Compute $T'_k(h, \ell)$ according to Equation (5.10);
        end
        $\left[S_h^k, S_\ell^k\right] \leftarrow$ Element with minimum $T'_k(h, \ell)$;
        Allocate stages $\left[S_h^k, S_\ell^k\right]$ into controller $M_{\mathrm{MCI}}^{\mathrm{first}}$;
        Update and re-order table $MCI$;
        Update and re-order table $RB$;
        Remove rows of allocated stages from table $SSB$;
        Update parameters that represent the first row of each table;

        return Information about the allocated stages;
    end
end
```

Chapter 5.2 and amounts to

$$O\left(\max\left\{QVN_{\max}^2, Q^2V^2K\right\}\right).$$

The time complexity of Phase 2 is $O\left(N_{\max}K\right)$ and the time complexity of Phase 3 is $O\left(\max\left\{N_{\max}^2, Q^2\right\}\right)$. Given that Phase 2 and 3 can be executed up to $N_{\max}K$ times, the total time complexity for MOMA is

$$O\left(\max\left\{QVN_{\max}^2, Q^2V^2K, N_{\max}^3K, N_{\max}Q^2K\right\}\right).$$

5.5 Summary of System-controlled Resource Allocation

To summarize, centralized and distributed resource allocation methods have been proposed in this chapter. Starting from the optimal CeRA method (Section 5.2), the distributed, hierarchical DiRA method (Section 5.3) shows how the optimum can be traded for a high degree of scalability. As a next step, MOMA (Section 5.4) extends the CeRA method heuristically to avoid a saturation of memory controllers whenever possible, and to minimize it in other cases. This is important in many-core systems with multiple memory islands when many memory-intensive applications are running concurrently.

These methods allocate resources in a system-controlled manner as they employ a single (CeRA, MOMA) or multiple (DiRA) controlling instances. This allows them to utilize global DiRA departs from centralized concepts and trades the optimum for a high degree of scalability. In the next chapter, methods for self-organizing software pipelines are proposed that completely avoid any controlling instances.

Chapter 6

Self-organizing Resource Allocation

6.1 Properties and Benefits of Self-organization

Self-organization is an approach to cope with the growing complexity of a system and is an aspect of autonomous computing [59]. In a self-organizing system, the entities that it comprises are responsible to self-optimize, self-configure, self-manage, or self-heal [41]. These properties are commonly referred to as the so-called *self-x* properties and form a key concept of Organic Computing [85, 103].

Self-organizing systems hence prefer distributed control that is based on social interaction over controlling instances. Their entities are responsible to observe the properties of their environment that are relevant to them and to communicate and interact with their peers in order to achieve their goals [45]. Following their reference model proposed by IBM, self-organizing entities operate in a control loop of monitoring, analyzing, planning and execution (MAPE) [41].

Key benefits of self-organizing systems include that they are highly scalable, resilient, and adaptive to changes [45]. The concept of self-organization is successfully employed in many instances. The authors of [44] present a comprehensive survey.

In the following, two concepts for self-organizing software pipelines are proposed to transfer those benefits to resource allocation in many-core systems.

First, the novel concept of Pipelets is introduced in Section 6.2. Pipelets allow software pipelines to optimize their resource allocations at runtime based on observations and interactions with other Pipelets. This way, they use local interactions to improve the system throughput.

In a second step, Section 6.3 applies the self-organizing resource allocation introduced by Pipelets to the DiRA method (Section 5.3). This way, the benefits of system-controlled resource allocation can be combined with the benefits of self-organization. We show how multiple hierarchical controlling instances (i.e. clusters) can act autonomously to increase resilience against unreliable hardware.

6.2 Pipelets

This section shows how self-organization can be used to adapt resource allocations at runtime based on local observations and interactions. To achieve this, *Pipelets* are proposed as self-organizing stages of software pipelines. They adapt resource allocations at runtime by migrating among cores in order to increase the performance of a system by balancing its load. The concept of Pipelets has been published in [52].

6.2.1 Definition of Pipelets

A Pipelet is a task forming one stage of a software pipeline (see Figure 6.1) with the following properties:

- It can adapt resource allocations by migrating between cores at runtime.
- It interacts with other Pipelets.
- Pipelets aim at optimizing their application's performance when an established resource allocation becomes inefficient.

In the following, the term *bottleneck* will be used to express the slowest stage of a software pipeline. Such a bottleneck limits its applications performance. Pipelets detect bottlenecks and take action to resolve them when possible using the means described in the following section.

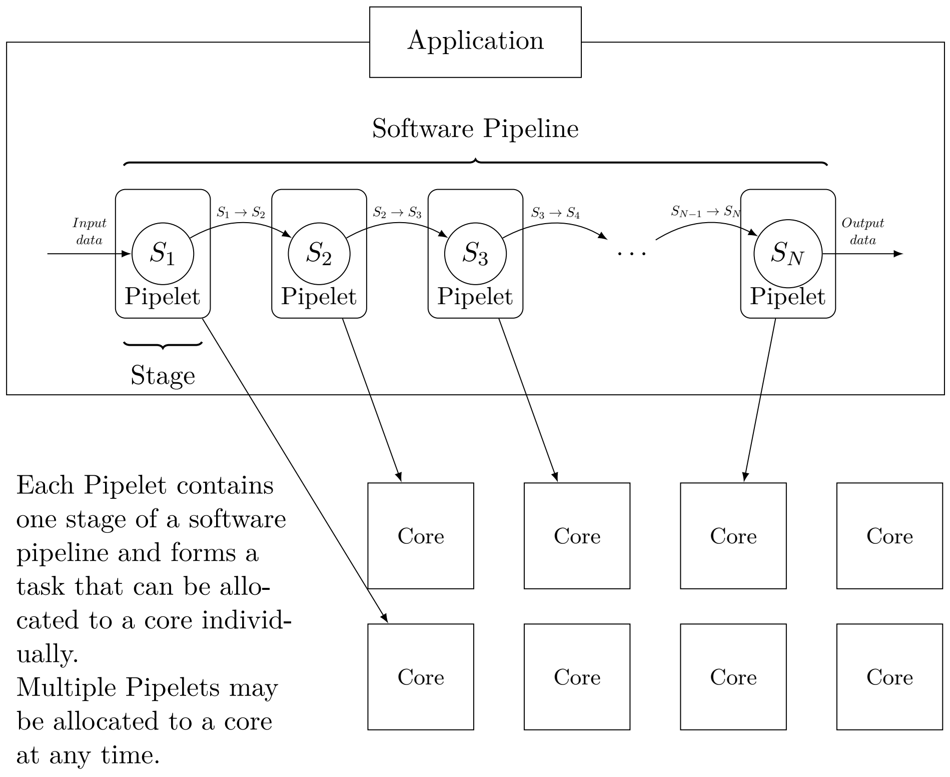
6.2.2 Actions and Phases

The means of self-organization are:

- *Bottleneck Relief* to improve the throughput when a bottleneck is detected, and
- *Contraction* to reduce the communication distance (i.e. number of hops) between them to reduce the bandwidth requirements.

As Pipelets do not dispose of information about the global system state, they *interact* (Section 6.2.3). In the following, we detail how Pipelets achieve self-organization and how they interact.

Pipelets *exploit* properties of software pipelines to achieve self-organization. To achieve self-organization, a Pipelet must be able to (a) find out if it limits the throughput of the application it belongs to, and to (b) adapt



Each Pipelet contains one stage of a software pipeline and forms a task that can be allocated to a core individually.
Multiple Pipelets may be allocated to a core at any time.

Figure 6.1: Relationship of Pipelets to Applications, Tasks and Stages

resource allocation by migrating to another core in a way that improves the throughput by balancing the load.

Therefore, a Pipelet must be able to estimate the impact of adapting resource allocations so it can choose one that improves the throughput of its application without impairing the throughput of other applications. Both (a) and (b) can be accomplished when Pipelets exploit the following prop-

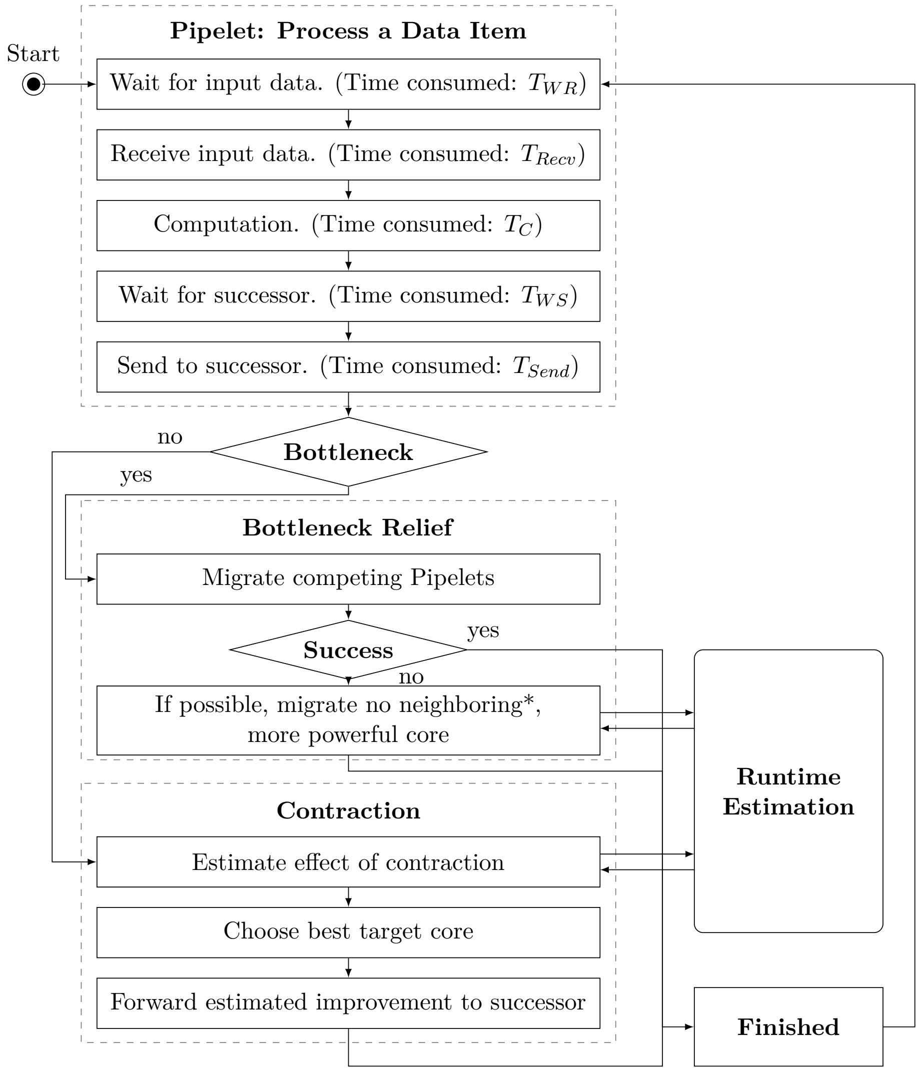

Pipelet: Process a Data Item
Start
Wait for input data. (Time consumed: T_{WR})
Receive input data. (Time consumed: T_{Recv})
Computation. (Time consumed: T_C)
Wait for successor. (Time consumed: T_{WS})
Send to successor. (Time consumed: T_{Send})
Bottleneck
no
yes
Bottleneck Relief
Migrate competing Pipelets
Success
yes
no
If possible, migrate no neighboring*, more powerful core
Runtime Estimation
Contraction
Estimate effect of contraction
Choose best target core
Forward estimated improvement to successor
Finished

Figure 6.2: Overview of Pipelet self-organization

erties of software pipelines:

(1) Each stage repeatedly processes data items in the following steps: first, it waits for and receives input data, followed by computation, and then it passes the output to its successor as soon as this is ready to receive it.

(2) The slowest stage of a pipeline limits its throughput (i.e. it causes a bottleneck).

(3) Preceding and succeeding stages of a software pipeline communicate once after a data item is processed, passing the output data from the *predecessor* as the input data of its direct *successor*. There is no other communication.

(4) The peak memory requirement of pipeline stages is often during computation because many buffers are freed after passing the output to the successor.

Pipelets **exploit** these properties in the following way: The strict temporal execution pattern of (1) enables Pipelets to measure the different time phases (by repeatedly querying the system time) that are consumed for processing each data item: Figure 6.2 illustrates how T_{WR} and T_{Recv} denote the time required for waiting for and receiving the input data, while T_C denotes the time consumed by computation. Likewise, T_{WS} and T_{Send} denote the times for waiting for the successor and sending data to it. Algorithm 6 illustrates the main loop (including measurements of its time phases) of a Pipelet.

As the slowest stage of an application limits its throughput (2), other stages need to wait. Consequently, for an application k where P_k denotes the set of its Pipelets, we define λ_k as the time consumed for processing each data item:

$$\begin{aligned}\forall p, q \in P_k : T^p_{WR} + T^p_{Recv} + T^p_C + T^p_{WS} + T^p_{Send} = \\ T^q_{WR} + T^q_{Recv} + T^q_C + T^q_{WS} + T^q_{Send} = \lambda_k\end{aligned} \tag{6.1}$$

The one-to-one communication restriction of (3) reduces the potential impact that migrating a Pipelet can have to the throughput of other Pipelets. Only Pipelets that are allocated to the target core may be impacted.

Additionally, we define the *slack* of a Pipelet p as the time it needs to wait for its predecessor and successor:

$$slack_p = T^p_{WR} + T^p_{WS} \tag{6.2}$$

The Pipelet that causes a bottleneck has a *slack* of 0. The throughput F_k of an application k is defined as:

$$F_k = \frac{1}{\lambda_k} \tag{6.3}$$

To increase F_k, the Pipelet b_k causing the bottleneck of k tries to reduce T^b_{Recv}, T^b_C, or T^b_{Send} to decrease λ_k. The rest of this section details in which way this can be achieved.

The property (4) of software pipelines implies that the size of the task context (program code, stack, registers and heap) of a Pipelet is minimal directly after processing a data item (as it may deallocate any temporary buffers required for computation). To achieve a low task migration overhead, Pipelets migrate after processing a data item is completed. Figure 6.2 shows how Pipelets perform their main loop, i.e. a data item is processed, and the steps required for self-organization: After the output data has been sent to the successor, a Pipelet evaluates its *slack* to see if it causes a bottleneck. If so, it tries to resolve it as detailed in Section 6.2.2.

Bottleneck Relief

If a Pipelet p of application k causes a bottleneck, it may increase the throughput F_k by decreasing T_{Recv}, T_C or T_{Send}, thus decreasing λ_k for all Pipelets $p \in P_k$. Therefore, it asks other Pipelets that are allocated to its core to migrate in order to free resources. These Pipelets are in turn responsible to find possible target cores. If migrating other Pipelets would decrease (any of) their application's throughputs beyond the gain for k, this option is discarded and b itself tries to migrate to a different core in its *neighborhood*, which is a set of cores physically closest to it (the size of the neighborhood is a design parameter). Small neighborhood sizes might increase the number of task migrations (because it limits the search space for each decision), while large neighborhoods increase the overhead. In our experiments, we find a neighborhood size of 12 cores provides good results. If a Pipelet is not responsible for a bottleneck, it tries to *contract*.

Contraction

Pipelets which do not cause a bottleneck *contract*, i.e. they try to migrate to the spatial proximity of their predecessor if this does not impair throughput. The goal of contraction is to reduce communication volumes. This is desirable because it may potentially increase the performance of other Pipelets that require higher bandwidths. Contraction is performed as follows: each time a data item has been processed, each Pipelet p_i evaluates $\mathbb{D}_{p_i} = D_{p_i,p_{i-1}} + D_{p_i,p_{i+1}}$, which denotes the sum of the distance (e.g. the hop count, i.e. the number of hops between two cores.) to its predecessor and to its successor.

Algorithm 6: Pipelet main loop with time measurements

```
while application running do
    // The first Pipelet has no predecessor
    if Have Predecessor then
        $t_0$ = GetTime();
        WaitForData( Predecessor );
        $t_1$ = GetTime();
        $T_{WR} = t_1 - t_0$;
        inputData = ReceiveData( Predecessor );
        $T_{Recv}$ = GetTime() - $t_1$;
    end

    $t_2$ = GetTime();
    // Compute performs the computation of the Pipelet
    outputData = Compute( inputData );
    $T_C$ = GetTime() - $t_2$;

    // the last Pipelet has no successor
    if Has Successor then
        $t_3$ = GetTime();
        WaitForRecipient( Successor );
        $t_4$ = GetTime();
        $T_{WS} = t_4 - t_3$;
        SendData( Successor, outputData );
        $t_5$ = GetTime();
        $T_{Send} = t_5 - t_4$;
    end

    Interact with other Pipelets (Section 6.2.3);
end
```

It receives I_{p*} that denotes the improvement (reduction) in $\mathbb{D}$ for one of its predecessors $p*$ (the first stage does not receive a value), and estimates the impact on its throughput for every possible migration to a core in its neighborhood that would decrease $\mathbb{D}_p$ without impacting any application's throughput. If it finds that its decrease in $\mathbb{D}_p$ exceeds I_p*, it updates this value and sets $p*$ to p. Next, it forwards I_{p*} to its successor. The Pipelet without a successor informs the Pipelet $p*$ with the largest positive improvement $I_{p*} > 0$ to perform the corresponding task migration. No contraction is carried out while a Pipelet adapts resource allocations to relieve a bottleneck.

6.2.3 Interactions

Pipelets interact to achieve self-organization. Figure 6.3 shows a sequence diagram of the interactions for a Pipelet p_i (p_{i+1} is its successor, while o, r, and s are other Pipelets that may but need not belong to the same application).

Pipelets interact in three cases: 1) p_i interacts with p_{i+1} to send the output data. Secondly, if p_i is not causing a bottleneck, it interacts with other Pipelets in its neighborhood (in this example o, r and s) to perform contraction. Therefore, it requests runtime estimates for possible contractions as described in Section 6.2.2. Thirdly, if p_i causes a bottleneck, it interacts with the other Pipelets in its neighborhood to estimate which of the possible migrations to relieve the bottleneck (as described in Section 6.2.2) offers the best improvement.

Pipelets interact with so-called *core guards*. Core guards are helper tasks that run on every core. They are responsible for (a) virtualizing communication between Pipelets to allow MPI communication orthogonal to their physical location, (b) replying to their status requests (CPU type and load, bandwidth usage, and a list of allocated Pipelets), and for (c) helping Pipelets migrate to its core. Figure 6.4 shows how Pipelets communicate via core guards. Pipelets receive status information such as the CPU load, the NoC bandwidth usage, and the list of Pipelets allocated to this core.

As a consequence, Pipelets interact in spatial (i.e. neighborhood) and temporal (i.e. predecessor/successor relationship) proximity. These limitations induce local, sub-optimal decisions, which trades scalability for a loss of optimality. However, their self-organizing behavior limits their overhead, which does not depend on the number of cores. Hence, Pipelets are scalable for very large systems.

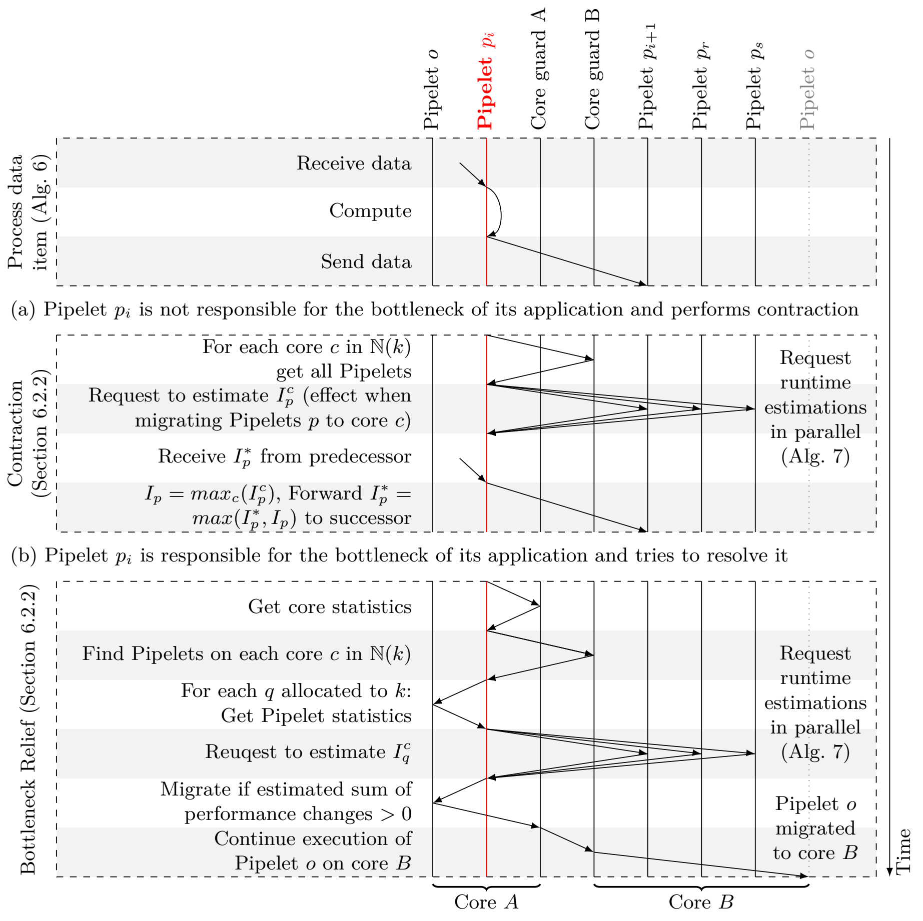

Pipelet o
Pipelet p_i
Core guard A
Core guard B
Pipelet p_{i+1}
Pipelet p_r
Pipelet p_s
Pipelet o
Process data item (Alg. 6)
Receive data
Compute
Send data
(a) Pipelet p_i is not responsible for the bottleneck of its application and performs contraction
Contraction (Section 6.2.2)
For each core c in $\mathbb{N}(k)$ get all Pipelets
Request to estimate I_p^c (effect when migrating Pipelets p to core c)
Receive I_p^* from predecessor
$I_p = max_c(I_p^c)$, Forward $I_p^* = max(I_p^*, I_p)$ to successor
Request runtime estimations in parallel (Alg. 7)
(b) Pipelet p_i is responsible for the bottleneck of its application and tries to resolve it
Bottleneck Relief (Section 6.2.2)
Get core statistics
Find Pipelets on each core c in $\mathbb{N}(k)$
For each q allocated to k: Get Pipelet statistics
Reuqest to estimate I_q^c
Migrate if estimated sum of performance changes > 0
Continue execution of Pipelet o on core B
Request runtime estimations in parallel (Alg. 7)
Pipelet o migrated to core B
Time
Core A
Core B

Figure 6.3: An example of the interaction between Pipelets in several scenarios. At the top, a data item is processed by Pipelet p_i. In the middle, (a) shows the interactions that take place when Pipelet p_i is not responsible for a bottleneck. After another data item is received, (b) shows the interactions that p_i initiates when it is responsible for a bottleneck. Ultimately, another Pipelet o is migrated from core A to core B, which relieves the bottleneck of Pipelet p_i on core A.

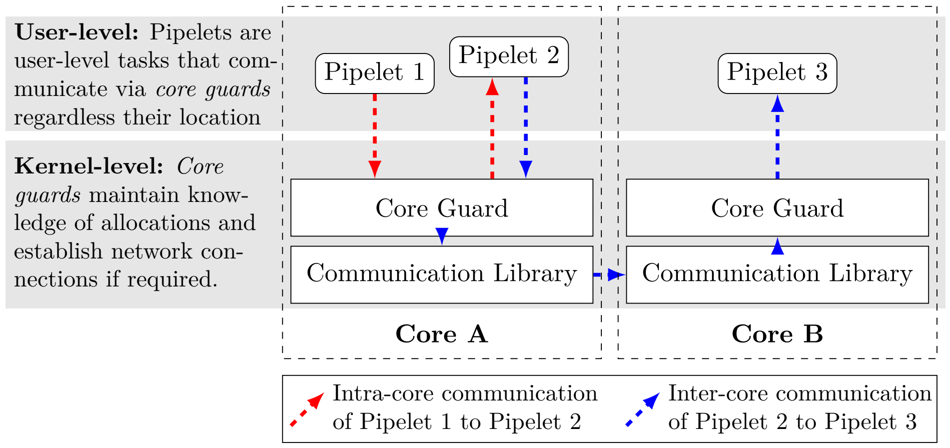

User-level: Pipelets are user-level tasks that communicate via *core guards* regardless their location
Kernel-level: *Core guards* maintain knowledge of allocations and establish network connections if required.
Pipelet 1
Pipelet 2
Pipelet 3
Core Guard
Core Guard
Communication Library
Communication Library
Core A
Core B
Intra-core communication of Pipelet 1 to Pipelet 2
Inter-core communication of Pipelet 2 to Pipelet 3

Figure 6.4: Pipelets communicate via so-called core guards

6.2.4 Runtime Estimation

Algorithm 7 shows how the runtime estimator of each Pipelet calculates the impact of a potential migration on T_C, T_{Recv}, and T_{Send}: The input parameters are the requirements of the Pipelet and the direction of the migration (i.e. either migrating *to* or *away from* the core, i.e. requiring or giving up resources). Timing estimations are defined as the product of the measured timings (T_C, T_{Recv} and T_{Send}) and the percental changes in resource availability (as required or given up by the Pipelet that is potentially migrated). When T_{Recv} or T_C increase, the *slack* decreases until it reaches 0.

6.3 Self-organizing Software Pipelines

The previous section has shown how self-organization can effectively and efficiently adapt resource allocations at runtime. This way, a system can respond adaptively to unpredictably starting or stopping of applications and even to drastic changes in the resource requirements of running tasks without controlling instances. However, while self-organization allows for distributed, robust methods, a limitation to local observations can lead to sub-optimal results as compared to system-controlled methods. In the following, we show how the concept of self-organizing Pipelets can be combined with the system-controlled DiRA method of Section 5.3 to combine the benefits of both approaches. This way, self-organizing software pipelines can

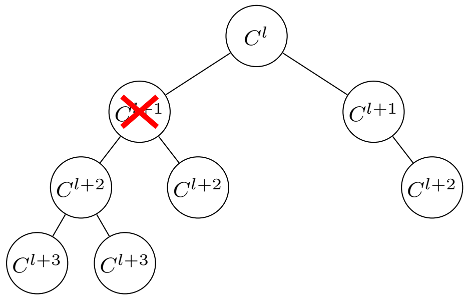
Algorithm 7: Runtime estimation

```
Input:
  $Dir : boolean$            Direction: migrating to or away from core
  $CPU, BW_{in}, BW_{out}$   Resource requirements (delta)
Definitions:
  $t_{new}$   Type of new (evaluated) core
  $t_{old}$   Type of old (current) core
Result:
  $I_p$   Performance improvement for $p$ (can be negative)

if $Dir == away\ from\ core$ then
    // Invert requirements (free resources)
    $\{CPU, BW_{in}, BW_{out}\} = -1 * \{CPU, BW_{in}, BW_{out}\}$;
end

$\widetilde{T_C} = T_C * (1 + CPU)$;
$\widetilde{T_{Recv}} = T_{Recv} * (1 + BW_{in})$;
$\widetilde{T_{Send}} = T_{Send} * (1 + BW_{out})$;

return $I_p = T_{Recv} - \widetilde{T_{Recv}} + T_C - \widetilde{T_C} + T_{Send} - \widetilde{T_{Send}}$;
```

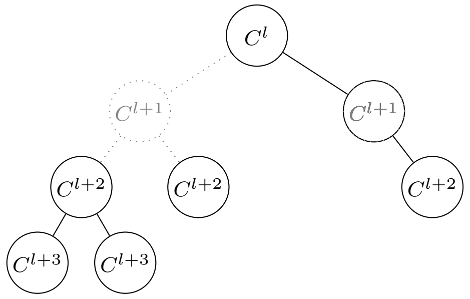
allocate resources in a way that is resilient to unreliable hardware, while near-optimal allocations can be achieved.

When the reliability of a system cannot be guaranteed and cores may *malfunction* (i.e. they stop to work correctly temporarily or permanently), the tasks allocated to them are interrupted. In the following, we illustrate how the problems caused by unpredictably malfunctioning cores can lead to a significantly decreased system throughput and how the *resilience* of our distributed method can be increased. For simplicity, we use the term *cluster* in the following to express the task that implements the functionality of a cluster of the proposed DiRA method of Section 5.3.

When a cluster is allocated to a malfunctioning core, the integrity of the hierarchical tree structure of clusters of Section 5.3 is corrupted. Figure 6.5 shows an example how interrupting a cluster corrupts the integrity of their hierarchical structure (a), and how this leads to disconnected sub-trees, as shown in (b).



(a) The hierarchical tree of DiRA. When a core malfunctions and a cluster is allocated to this core, its task is interrupted and the tree is disconnected.



(b) The DiRA resource allocation can adapt allocations for the disconnected sub-trees, but it cannot re-allocate resources between applications that belong to different sub-trees.

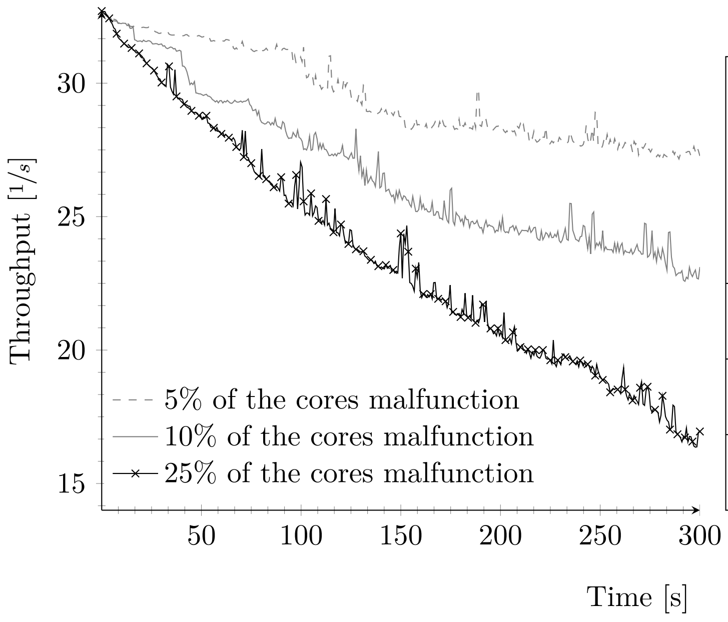
Figure 6.5: Effect of malfunctioning cores that corrupt the integrity of the hierarchical tree structure of our DiRA resource allocation method: a malfunctioning core "removes" a cluster and thus the remaining sub-trees are disconnected. Consequently, resources be re-allocated among them (i.e. no "exchange" of resources is possible between sub-trees).

As a result, our distributed method is still able to (re-)allocate cores in the remaining sub-trees. However, there is no exchange of cores among sub-trees. To illustrate how this may lead to a significantly decreased system throughput, let us consider a case of two disconnected sub-trees A and B, and each sub-tree contains only one application, a in A and b in B. When the resource requirements of a increase and the requirements of b remain the same (or decrease), giving cores from B to A would increase the system throughput. However, as A and B are disconnected, this is not possible. This is also the case for more than one application per sub-tree, and for more than two sub-trees. Thus, disconnected sub-trees may lead to core distributions that result in a decreased throughput. Simulations of a 1024-core system where cores malfunction randomly during a 300-second interval show that interrupting clusters can result in significantly reduced system throughputs of 17%, 29% and 50% for a malfunctioning of 5%, 10% and 25% of the cores, respectively, as shown in Figure 6.6. To address this problem, we propose to employ self-organization to restore the integrity of the hierarchical structure of clusters even when cores malfunction. This self-organization of clusters is characterized as follows:

- A cluster detects when their parent cluster interrupted (e.g. when it does not respond).

- When its parent was interrupted, a cluster searches for another cluster (that is not among its children) by sending a connection request to randomly selected cores.
- Once such a cluster is found, both clusters join their sub-trees.
- This way, the integrity of the hierarchical tree is restored.

Algorithm 8 shows how this can be achieved. When a cluster h finds that its parent was interrupted, it starts to resolve this in parallel to its duties of allocating cores among its children. Until all cores have been searched or a new parent is found, the cluster tries to establish a connection to a random core. If a cluster h' is allocated to this core and h' is not among its children, it *joins* h', i.e. h' lists h among its children, and h sets h' as its new parent. When this is achieved, it stops searching for a new parent. A handshake



Throughput reduction	Core malfunction rate
17%	5%
29%	10%
50%	25%

Figure 6.6: Effect when 5%, 10%, and 25% of cores malfunction over a period of 300 seconds in a 1024-core system running 275 applications. It can be observed that the resulting disconnected sub-trees lead to significantly decreasing throughput.

Algorithm 8: Self-Organization to Restore the Integrity of the Hierarchical Structure of Clusters

```
Input:  Cluster $h$ with malfunctioning parent
Result:  Restored hierarchical tree

for each core $c$ do
    $c$ = Random core;
    if a cluster $h'$ is allocated to $c$ then
        if $h'$ does not contain $h$ among its children then
            set $h'$ as new parent;
        end
    end
end
```

protocol prevents corner-cases of two sub-trees that both search for a new parent to join each other.

This way, the integrity of the hierarchical structure of clusters our distributed method can be restored.

6.4 Summary of Self-organization

To summarize, self-organization can be employed for resource allocation so that a high degree of scalability and an increased resilience against failing cores can be achieved. When a self-organizing method does not employ controlling instances, local observations, interactions, and runtime estimations can guide the decisions to adapt resource allocations at runtime. Local decisions can lead to suboptimal results and hence, optimal solutions may hardly be achieved. While our Pipelets cannot provide globally optimal resource allocation, the experimental results show that deploying Pipelets results in a high system throughput without controlling instances.

Chapter 7

Task Migration

Re-allocating resources at runtime may require to migrate tasks between cores and hence, task migration is an important cornerstone for the resource allocation methods proposed in this thesis. As the overhead of task migration can be significant, an efficient task migration mechanism is key to quickly and successfully re-allocate resources at runtime with minimal overhead. However, the state-of-the-art task migration mechanisms do not sufficiently address the requirements of adapting resource allocations in many-core systems: Application-level task migration mechanisms require careful implementation in a way that is specific to each application [2, 13, 20, 80, 90]. The resulting development effort may be large, time-consuming, and error prone and hence, such strategies are hardly applicable for the large variety of complex, user-centric applications found in state-of-the-art embedded many-core systems. System-level migration mechanisms focus on unreliable and potentially insecure networks when transferring tasks, require binary translation of data, or induce significant overhead which is not suitable for migrating tasks frequently [4, 10, 100].

To overcome these drawbacks, this thesis proposes a novel system-level mechanism for task migration, CARAT [53]. CARAT migrates tasks transparently and exploits the behavior of tasks in order to minimize the performance overhead of task migrations. Furthermore, CARAT adapts its transfer policy at runtime based on the observed behavior of the task. This allows frequent migrations with a low impact on system performance.

Migrating a task between cores that do not share their memory requires to transfer the *task context*, i.e. its entire data. Table 7.1 shows how the task context consists of the program code, stack and register contents, and the heap memory of a task. Typically, the program code is placed in read-only memory and does not change once a task has been started. Thus, it can be transferred before the task is stopped on the source core. However, stack and register contents as well as the heap memory may change frequently at runtime. Heap memory may be large and hence, its transfer may consume considerable time. When CARAT and many other state-of-the-art task migration mechanisms migrate a task, the task may be resumed on the destination core after its program code, stack, and register contents have been transferred but before its heap memory has been transferred completely. Heap memory is commonly transferred on the granularity of memory pages. When the task tries to access data that has not been transferred, a so-called *page fault* exception is raised and the task execution is paused until the system finds that it can be resumed. After the missing page of this data has be transferred, the task can thus be resumed.

The performance overhead of a task migration is largely driven by this waiting time for pages of heap memory that are accessed on the destination but have not yet been transferred. Thus, achieving a low performance overhead requires to carefully choose the order in which to transfer the memory pages. This order should match the order of the accesses on the destination core as closely as possible.

Type	Change Frequency	Typical Size	Description
Program Code	rarely	few MB	Contains the executable processor instructions that form a program
Stack Memory	very frequently	up to 1 MB	Stores local variables, function arguments, and return addresses
Registers	very frequently	few KB	Stores the operands and results of processor instructions
Heap Memory	frequently	KB to GB	Stores the working data of a task

Table 7.1: Description of the data that form a task context.

The state-of-the-art task migration mechanisms propose different policies to transfer the task context [4, 100]: Most importantly, the *lazy-copy* policy transfers pages only when they are accessed and a page fault occurs. The *pre-copy* policy transfers the entire context once a task is still running at the source core and monitors changes to data that has already been transferred. Hence, some pages have to re-sent once the task is stopped on the source core. The *post-copy* policy only transfers program code and the contents of the stack and registers before the task is resumed on the destination core and sends the heap memory in parallel to the continued execution on the destination core.

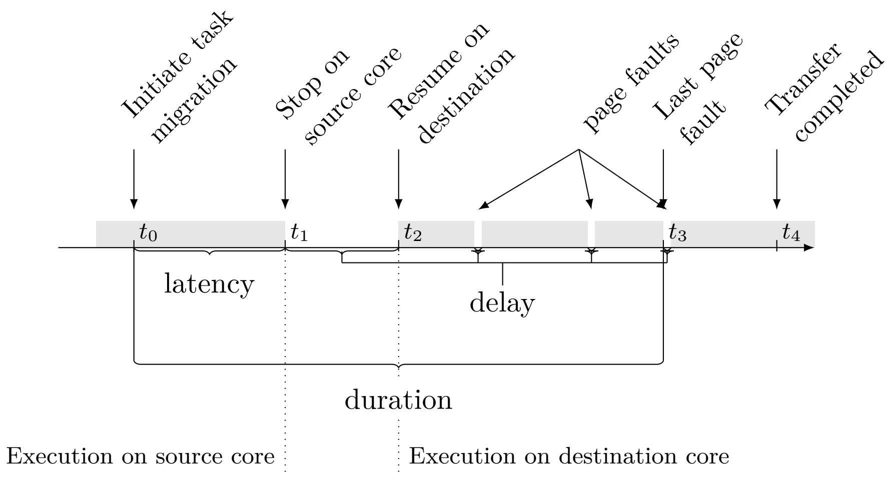


Figure 7.1: Latency, duration and delay of task migration

To analyze the different properties of a task migration mechanism, the time consumed by task migration can be divided into multiple phases.

- The *latency* denotes the timespan between initiating task migration and stopping it on the source core.
- The *delay* denotes the time while the task is paused. This consists of the time between stopping the task on the source core and resuming the task on the destination and the time consumed by page faults.
- The *duration* of a migration denotes the time from initiating the task migration to the last page fault that occurs on the destination core.

The relation of latency, duration, and delay is depicted in Figure 7.1. The total amount of data (in MB) that is transferred for migrating a task constitutes the communication overhead.

The performance penalty of migrating tasks for (re-)allocating resources corresponds to the migration delay. Hence, a migration mechanism should aim at reducing this delay as much as possible. In order to derive a policy, the memory access behavior of diverse applications is analyzed in the following section.

7.1 Memory Access Behavior Analysis

The transfer policy which guides the sequence in which memory pages are transferred should match the memory access behavior of the task closely. Mismatches between predicted and actual behavior can result in page faults, which contribute to the delay. Furthermore, the transfer of data that is no longer used by the application unnecessarily increases the bandwidth requirements. To derive a good transfer policy, a complex, state-of-the-art

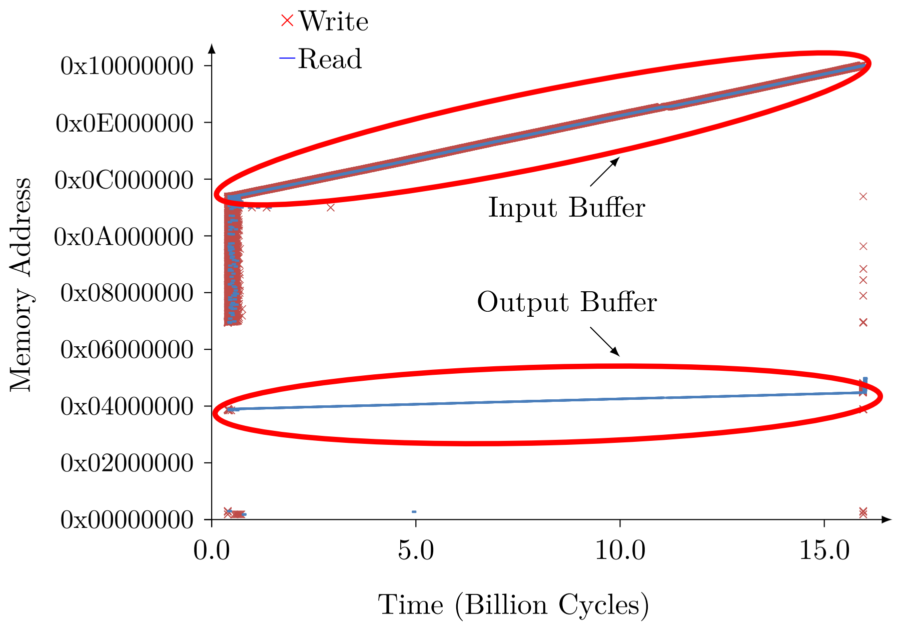


Figure 7.2: Memory access pattern of the “7Zip encoder” application (input: 85 MB binary file). To enhance the visibility, only the first accesses to each memory page are shown.

multimedia application “x264 encoder”, the 7Zip LZMA implementation “7Zip encoder” and a state-of-the-art embedded systems “robotic application” that performs pipelined stereo vision, feature extraction and stereo matching are analyzed in the following. Figure 7.2, 7.3, and 7.4 show the memory access behavior for these applications, respectively, after a randomly triggered task migration. The figures show only accesses to pages that have been allocated prior to the task migration as other pages are allocated on the destination core. More specifically, the figures show the first time a memory page is accessed after the task migration has been initiated. The reason for showing only the first accesses is that after the first access, the corresponding memory page must be available on the destination as otherwise, the task could not have been resumed.

A first observation shows that these applications have a very different memory access behavior, ranging from a linear behavior of the “7Zip encoder” application to a very random, irregular memory access of the “x264 encoder”.

From these observations, we gain the following insights: (a) Not all pages

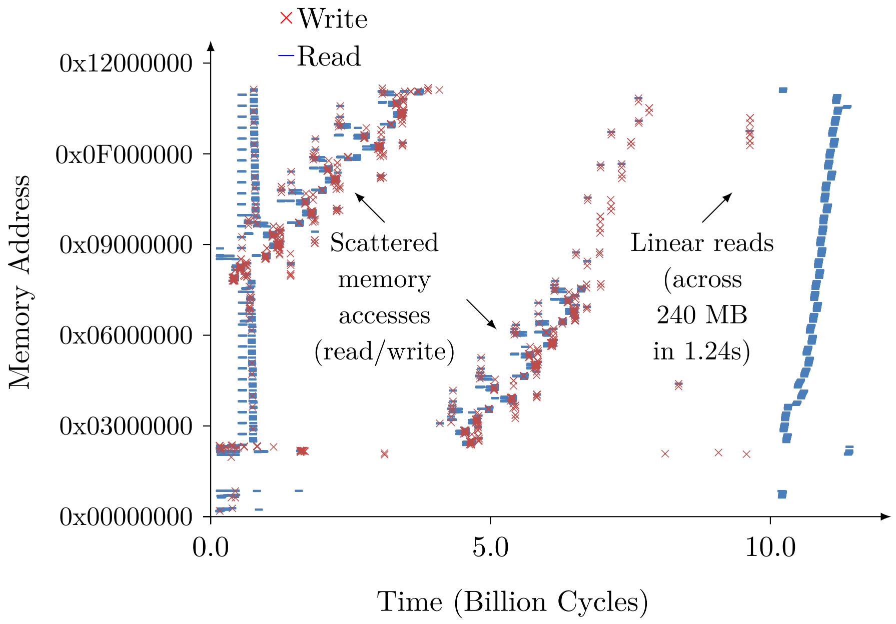


Figure 7.3: Memory access pattern of the “x264 encoder” application (input: 96 MB YUV movie in CIF resolution). To enhance the visibility, only the first accesses to each memory page are shown.

are required at the same time, and (b) when a page is accessed, it is *likely* that succeeding pages will be accessed soon. (c) Applications might separate read- and write buffers and thus, buffers that have predominantly been written to are unlikely to be accessed with heavy read accesses in the near future. (d) Some applications use small buffers to store frequently used operands, such as filter kernels, that may be accessed excessively. (e) Memory blocks are mostly accessed from front to back, not in reverse order. (f) Memory access behavior differs largely across applications and thus, a task migration mechanism should not suffer from unexpected accesses beyond the performance of the existing mechanisms and should not be tied to a specific application if a broad applicability is desired.

As the memory access behaviors of tasks may differ significantly, runtime adaptivity is needed.

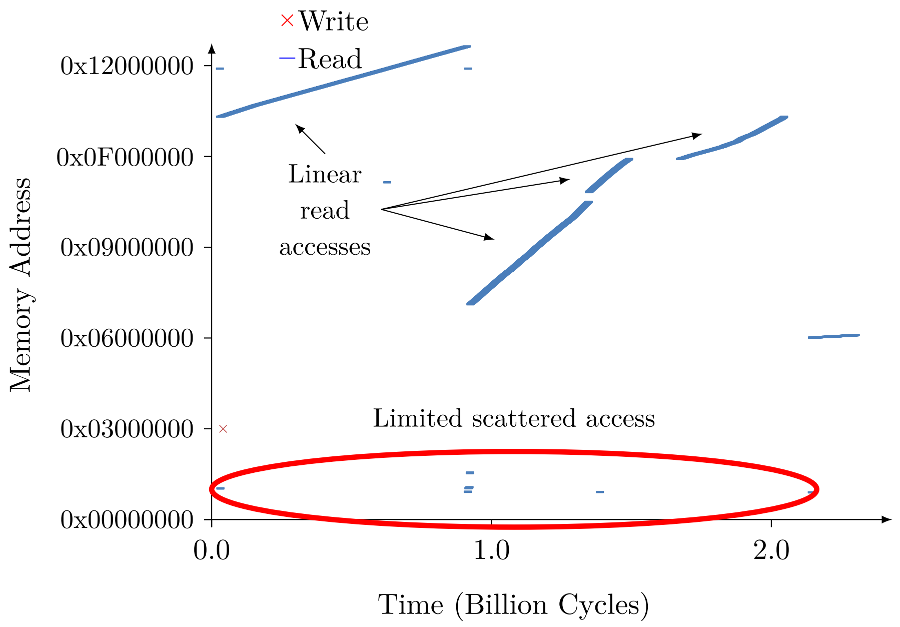


Figure 7.4: Memory access pattern of the "robotic" application (input: 640x480 stereo camera video sequence at 25 frames/second). To enhance the visibility, only the first accesses to each memory page are shown.

7.2 Runtime Adaptivity

CARAT achieves runtime adaptivity in two ways:

Runtime monitoring: CARAT monitors accesses to unavailable memory in a so-called Page Fault Queue (PFQ) as a basis to prioritize page transfer. The PFQ tracks the number of page faults that have occurred for each *memory block*. A *memory block* is a region of memory that consists of one or more consecutive pages that have been allocated at once. Prioritizing the transfer of frequently-used memory blocks exploits data locality and aims at avoiding future page faults. A state diagram for the transfer policy of CARAT is shown in Figure 7.5.

Tradeoff parameter α: A tradeoff parameter $\alpha \in [0, 1]$. α balances between a low latency or low bandwidth requirements. For $\alpha = 1$, CARAT tries to reduce the latency as much as possible, while smaller values steadily increase the latency but decrease the involved bandwidth requirements. CARAT converges to the *lazy-copy* mechanism for $\alpha = 0$ as no pages are transferred without being requested through a *page fault*. To achieve this behavior, each page transfer that is not a response to a *page fault* is scaled (multiplied) by α, while the small buffer advance transfers in the adapted *pre-copy* phase are scaled with $\sqrt{\alpha}$ because small buffer transfers cause relatively low communication traffic while delivering a high probability of reducing *page faults*. However, $\alpha = 1$ does *not* imply excessive bandwidth requirements; they are typically smaller than those of the *pre-copy* and *post-copy* mechanisms while delivering a significantly reduced latency.

7.3 Migration Policies

Based on the memory access behavior analysis and on the runtime adaptivity, the CARAT mechanism policies have been derived. CARAT combines a pre-copy and a post-copy phase, where selected memory pages are sent in parallel to the execution of the task on the source core. This continues until all selected pages have been transferred or until the maximum task migration latency is reached (latency threshold), or until a maximum amount of pages is transferred (bandwidth threshold). These thresholds are design-time parameters N, S_{Max}, and T_{Th}:

Algorithm 9: CARAT Transfer Algorithm: Phases 0 & 1

```
Input:
    $\alpha$              Latency/bandwidth tradeoff parameter
    $S_{Max}$             Maximum number of pages in pre-copy phase
    $Th_{Delay}$          Maximum delay threshold
Definitions:
    $T$                   Current time / cycle counter
    $\#S$                 Number of transferred pages
    $S_B$                 Pages to transfer from block $B$
    $Block(X)$            Block containing page $X$
    $\Gamma_B$            Largest page address of block $B$
    $PF_B$                Page fault counter for block $B$
    $PFQ$                 Page-fault queue
    $Next(X)$             Next page to send for page fault at $X$
    $N_1, N_2, N_3$       Design-time parameters (page-buffer size)
Phase 0: Initialization
Transfer program code;
Phase 1: Adaptive pre-copy phase
while $T < Th_{Delay}$ $AND$ $\#S < S_{Max}$ do
    $X \leftarrow$Choose next page$_\alpha$;
    Transfer $X$, break if $X = \emptyset$;
    $\#S \leftarrow \#S + 1$;
end
Phase 2: Switch execution
Pause task;
Transfer register contents and program stack;
Mark transferred but overwritten pages as missing;
// (Hardware support in the MMU is required)
Continue task execution on destination core;
```

Algorithm 10: CARAT Transfer Algorithm: Phase 3

```
***Phase 3:*** *Post-copy / page fault handler phase*
while  Task running do
    if  Page fault occurred at $X$ then
        // Page fault handler
        Cancel current transfer;
        $PFQ \leftarrow X \cup PFQ$;
        $PF_{Block(X)} \leftarrow PF_{Block(X)} + 1$;
        $Next(X) \leftarrow X$;
        if  $PF_{Block(X)} = \{1, 2, 3, > 3\}$ then
            $S_X \leftarrow \{X + N_1, X + N_2, X + N_3, \Gamma_{Block(X)}\}$;
        end
        //Scale upper transfer boundary with $\alpha$
        $S_X \leftarrow MAX(1,\ S_X \times \alpha)$;
    end

    for  each $Y \in PFQ$ do
        // Serve page fault queue (1-3 pages)
        // Interrupt if another page fault occurs
        for  $n = 0$ to $MIN(3,\ PF_{Block(Y)})$ do
            if $Next(Y) > S_Y$ then
                $PFQ \leftarrow PFQ \cap Y$;
                Exit For Loop;
            else
                Transfer $Next(Y)$;
                $Next(Y) \leftarrow Next(Y) + 1$;
            end
        end
    end
    for  each $\alpha\times$ Single-page blocks not yet sent do
        // Send remaining single-page blocks
        Send next single-page block $\alpha$;
    end
end
```

N denotes the maximum number of pages that should be transferred from the blocks that have been read from most recently at the time of initiating the task migration. This exploits the assumption that the block that has most recently been read from is likely to be used again soon.

S_{Max} denotes the maximum number of pages to be transferred in the pre-copy phase. This limits the maximum amount of data that is transferred prior to migrating the task, and hence limits the latency of task migration.

T_{Th} denotes the maximum latency of task migration.

For the design parameters N and B, we have chosen the values 8 and 4 arbitrarily. Optimal parameter values needs to be derived through design-space exploration, which is beyond the scope of this thesis.

Algorithms 9 and 10 show the pseudo-code for the three-phase policy:

Phase 1: Until the specified maximum migration delay is reached, do the following: First, send the $N \times \alpha$ pages that succeed the most recently accessed page of the B most recently read blocks. Second, send small buffers that have been read frequently, where the total number of buffers to transfer is multiplied by $\sqrt{\alpha}$. Third, send α (meaning $\alpha \times 100\%$) of the remaining pages that have not yet been accessed for read access in blocks where other pages have recently been read. Fourth, send α of the blocks that have been predominantly read after having been written.

Phase 2 pauses the task on the source core and transfers the stack and register contents. When this has been completed, the task is resumed on the destination core.

In *Phase 3*, the adaptive *post-copy* and *page fault* handler transfers selected pages in parallel to the resumed execution and handles page faults as described in the following: As the memory access behavior analysis shows that a fair fraction of allocated memory might not be used anymore (other than being freed) after the task migration, the adapted *post-copy* send handler does not send the entire memory of a task. This allows to reduce the total amount of bandwidth requirements. The unified *post-copy / page fault* handler transfers as follows: When a *page fault* occurs, the current transfer is canceled and the missing page is sent instantaneously. For the first *page fault* for this block, the missing page and the succeeding $N_1 \times \alpha$ pages are sent. For the second *page fault*, the succeeding $N_2 \times \alpha$ pages are sent and for the third fault, the succeeding $N_3 \times \alpha$ pages are sent. From the fourth *page fault* in a block, the entire block is transferred.

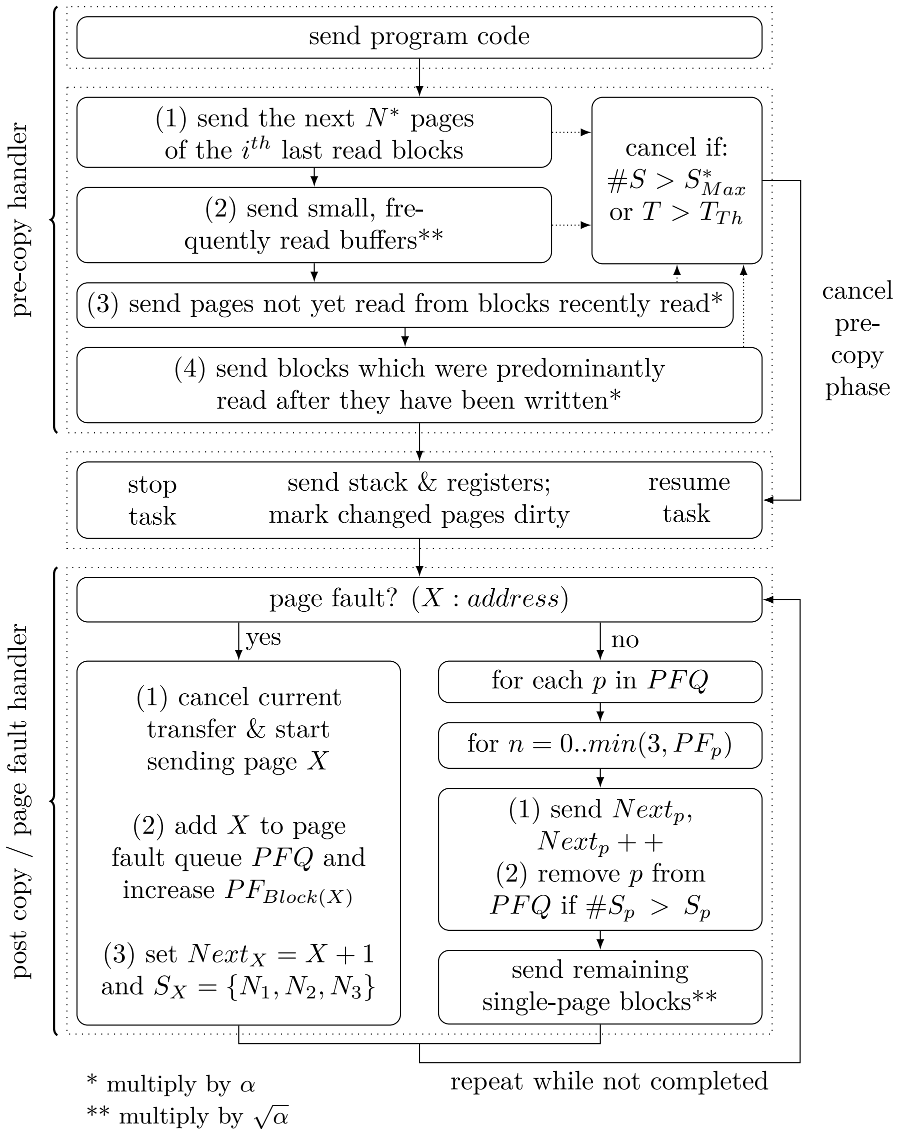

pre-copy handler
send program code
(1) send the next N^* pages of the i^{th} last read blocks
(2) send small, frequently read buffers**
(3) send pages not yet read from blocks recently read*
(4) send blocks which were predominantly read after they have been written*
cancel if: $\#S > S^*_{Max}$ or $T > T_{Th}$
cancel pre-copy phase
stop task
send stack & registers; mark changed pages dirty
resume task
post copy / page fault handler
page fault? (X : $address$)
yes
no
(1) cancel current transfer & start sending page X
(2) add X to page fault queue PFQ and increase $PF_{Block(X)}$
(3) set $Next_X = X + 1$ and $S_X = \{N_1, N_2, N_3\}$
for each p in PFQ
for $n = 0..min(3, PF_p)$
(1) send $Next_p$, $Next_p{+}{+}$
(2) remove p from PFQ if $\#S_p > S_p$
send remaining single-page blocks**
repeat while not completed
* multiply by α
** multiply by $\sqrt{\alpha}$

Figure 7.5: Schematic view of the CARAT algorithm. N, S_{Max}, and T_{Th} are design parameters.

When this prioritized transfer of missing pages is not yet completed before another *page fault* occurs in a different block, each new request is added to a queue which is served in a round-robin fashion, weighted with the number of *page faults* in each block to time-multiplex the communication with a per-page granularity. When the prioritized transfer has completed, single-page blocks are transferred. This accounts for the observation that applications may use a fair number of small blocks, which could increase the *page fault* count as the block-wise sending described in the *page fault* handler would fail to predict accesses to them.

A state diagram that illustrates these phases is depicted in Figure 7.5.

7.4 Exploiting Temporal Patterns

CARAT migrates task transparently, i.e. no support on application level is required. However, software-pipelined applications repeatedly process data items. While a data item is being processed, a stage may allocate buffers to store intermediate and output data, and may free these buffers once the data item has been processed. Figure 7.6 illustrates this pattern, where the

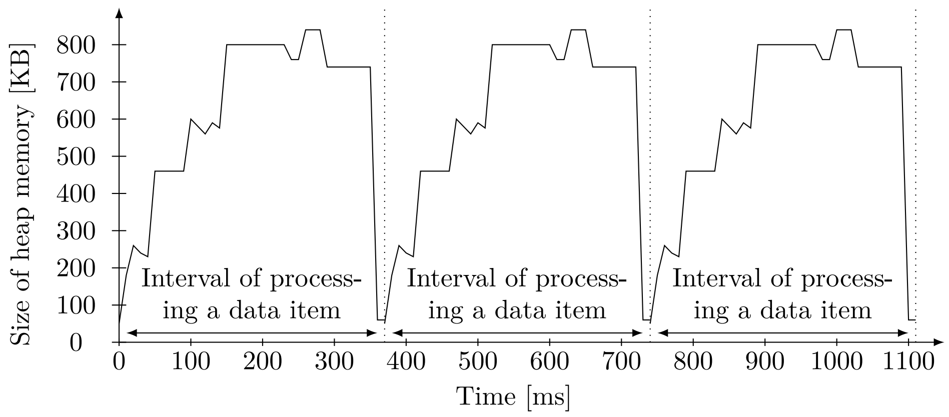


Figure 7.6: Size of the heap memory of a stage of a software pipeline. The stage processes three data items, while its heap memory size ranges from approx. 60 KB to approx. 840 KB. When CARAT migrates the task directly after a data item has been processed, the amount of heap memory that must be transferred can be reduced by approx. 93%.

heap memory size of a task ranges between approx. 60 and 840 KB. This temporal behavior of software pipelines can be exploited when adapting resource allocations by migrating tasks once a data item has been processed. To achieve this, the resource allocation methods proposed in this thesis expose an API to the tasks to notify the system-level resource allocator when a data item has been processed by a stage.

7.5 Summary of Task Migration

Based on a runtime-adaptive migration policy, the performance overhead of task migration can be greatly reduced. By combining transparent, system-level task migration with an application interface that allows to exploit the temporal behavior of software pipelines, frequent task migrations become feasible so that resources can be allocated and re-allocated at runtime. Hence, CARAT builds an efficient foundation for runtime resource allocation in many-core systems.

Chapter 8

Experiments and Evaluations

This chapter details the experiments conducted to evaluate the effectiveness of the methods proposed in this thesis and is organized as follows:

After discussing the experimental setup in Section 8.1, Section 8.2 explains the implementation of our methods on Intel's Single-Chip Cloud Computer (SCC) [43]. Section 8.3 explains the many-core system simulator used in the experiments. The benchmark applications that have been used for the experiments as well as their application scenario are detailed in Sections 8.4 and 8.5, respectively. To achieve a fair comparison, the state-of-the-art resource allocation baselines were adapted as described in Section 8.6. The performance of our methods is analyzed and compared to the state-of-the-art in Section 8.7, while their computational and communication overheads are discussed in Sections 8.8 and 8.9, respectively. Finally, Section 8.11 concludes this chapter.

8.1 Setup

Our experiments have been conducted on Intel's Single-Chip Cloud Computer (SCC) [43] and using a high-level many core simulator.

The SCC contains 48 cores which are connected with a Network-on-Chip (NoC). Its architecture is described in detail in Section 8.2.

Our high-level many core simulator executes task traces collected on the SCC and simulates the network-on-chip interconnect. The simulator delivers accurate information on the application- and system throughputs as well as on the communication volumes and overheads (algorithm runtimes have been collected on the SCC). It runs on a six-core AMD Opteron™8431 CPU (2.4 GHz) with 64 GB DDR3 RAM. The implementation of our simulator is described in Section 8.3.

The SCC allows measuring the computational overhead of our methods accurately, but as it integrates 48 cores, we cannot analyze the system throughputs and the communication overhead for larger systems. However, we measured the computational overhead on the SCC even for (virtually) large systems because these computations do not demand to dispose of the cores physically.

The following experiments have been conducted on the SCC:

- Computational overhead for up to 1024 cores.
- Throughput of the methods for up to 48 cores.
- Fusion/fission overheads.

The experiments conducted using our simulator include:

- Communication overhead.
- Throughput of our methods for systems with 128, 512 and 1024 cores.

8.2 Implementation on the SCC

Intel's Single-Chip Cloud Computer (SCC) integrates 48 x86 cores on a single chip [43] (45nm process, 1.3 billion transistors). The chip contains 24 tiles in 6 columns and 4 rows. Each tile contains two P54C cores, a network interface, a router, 16 KB Message Passing Buffer (MPB), two 256 KB L2 caches, and two cache controllers (CC), as shown in Figure 8.1. The Thermal Design Power (TDP) is 125W at 1 GHz (Mesh at 2 GHz). Access to off-chip DRAM (up to 32 GB) is facilitated via four off-chip memory controllers. Each core has a private address space of 1 GB, and memory access is maintained by an address Look Up Table (LUT). The virtual address space of each core is paged into 256 pages of 16 MB, and the LUT provides address translation and routing information. These LUTs can be programmed dynamically. There is no hardware support for cache coherence on the SCC. A configuration of the virtual address space of a core is shown in Figure 8.2.

A Network-on-Chip (NoC) connects the tiles with a 2D-mesh topology. The data links are 16 Bytes wide (plus a 2 Byte wide sideband channel) with a frequency of up to 2 GHz. The latency between hops is 4 cycles (2ns), and the bisection bandwidth of the NoC is 2 TB/s.

Message passing is facilitated via the Message Passing Buffers (MPB) on each tile. The size of the MPB coincides with the size of the L1 caches. Thus, MPB form a coherent, shared memory space that totals 384 KB for the entire chip.

The Single-Chip Cloud Computer runs a customized single-core Ubuntu-based Linux for the 48 individual cores. To boot a core, an image of the operating system is loaded into its virtual address space and then, registers are reset and execution starts at a defined address.

The Single-Chip Cloud Computer supports inter-core communication using sockets and via a proprietary communication infrastructure, RCCE.

To communicate via sockets, a (virtual) ethernet adapter (`RCKMB`) is exposed to the operating system. Libraries such as MPI build on top of this (virtual) ethernet adapter. This way, traditional communication protocols (such as TCP/IP) can be used for communication between cores.

RCCE is Intel's message passing Application Programming Interface (API) for the Single-Chip Cloud Computer and can be used with and without an operating system [113]. It uses the shared Message Passing Buffers (MPB) for communication. Accesses to the MPB are cached in L1 but bypass L2 caches. RCCE's programming model is a producer/consumer based, while a consumer has to poll the corresponding MPB for new data to arrive. RCCE was designed for use without operating systems and thus, its polling communication mechanisms can severely degrade the performance of multithreading operating systems.

8.3 Many-core System Simulator

Experiments for systems with a large number of cores are conducted using a high-level many-core system simulator. This simulator executes application traces that have been collected on Intel's Single-Chip Cloud Computer [43].

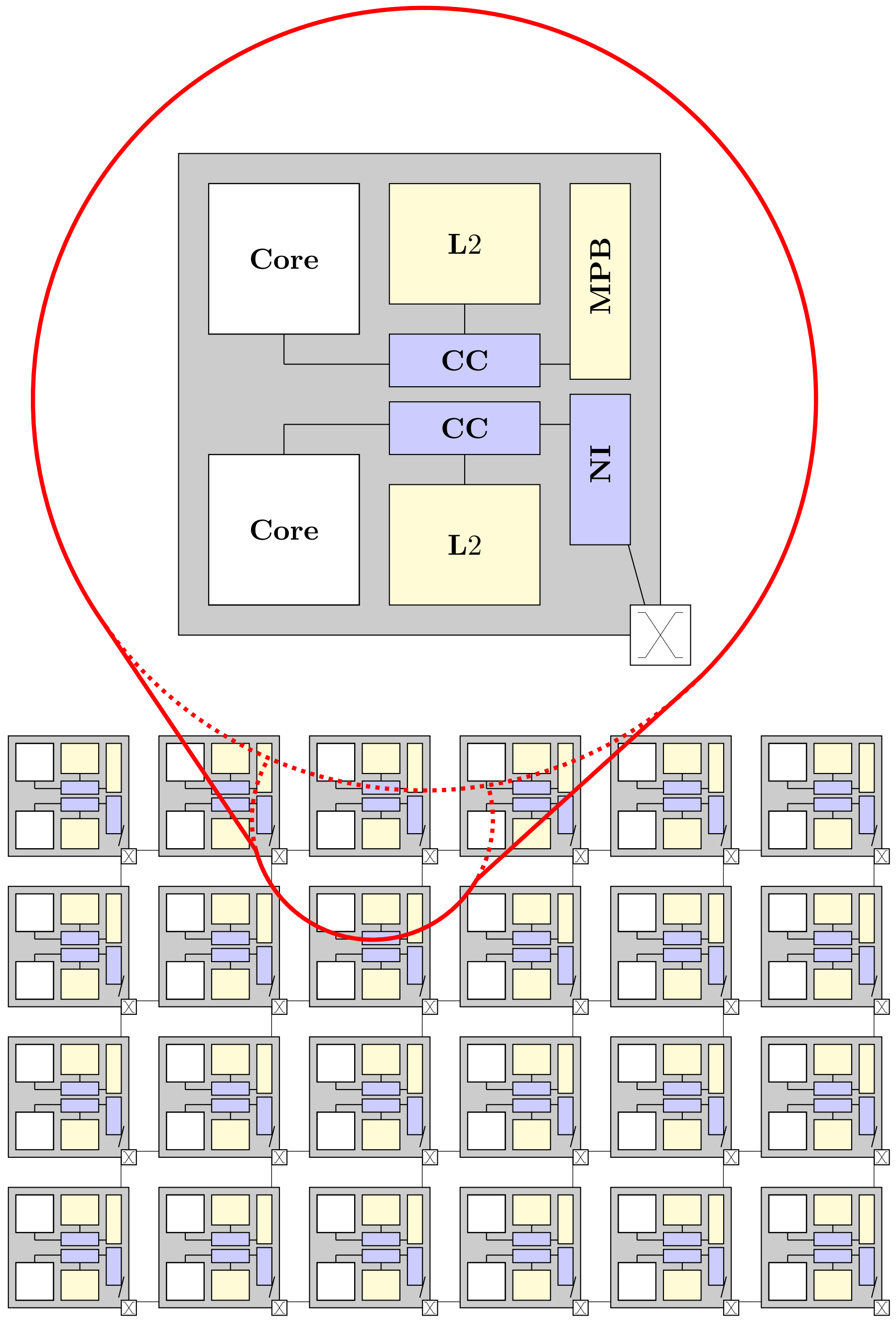


Figure 8.1: Schematic view of Intel's Single-Chip Cloud Computer

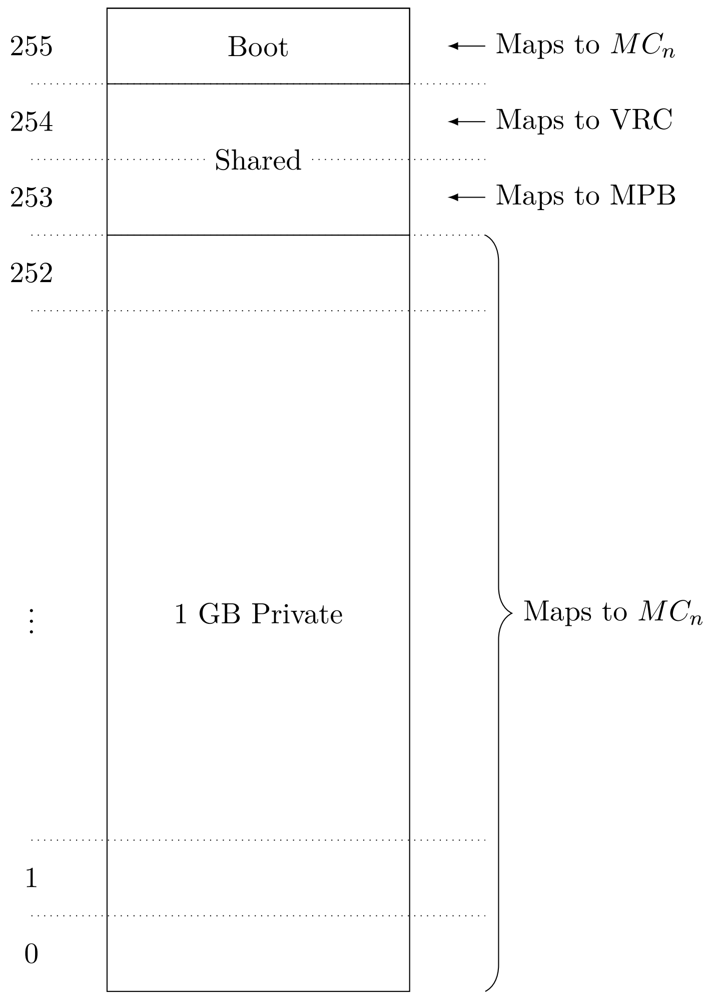


Figure 8.2: Address space layout of each core of the SCC. The “Boot” memory region as well as the 1 GB private memory map to the corresponding memory controller MC_n, while the shared memory region maps to the VRC and to the shared Message Passing Buffer (MPB).

The architecture that is simulated corresponds to Intel's Single-Chip Cloud Computer [43]: P54C cores connected via a 2D-mesh Network-on-Chip with X-Y routing and a link speed of 25 GiB/s with 2 cycles latency per hop. A more detailed description of this simulator can be found in Appendix A.

8.4 Benchmark Applications

For the experiments, a set of complex parallel real-world applications is used which is shown in Table 8.1.

The application "automotive" is a vision-based application that finds, tracks and highlights objects visually in a stream of stereo video data. It performs image enhancements and rectification followed by Harris Corner Detection and Scale-Invariant Feature Transform (SIFT). Isopolar-geometry-based stereo matching is used to calculate the depth information of interesting points, while a pattern matching algorithm allows to recognize and track objects and to highlight them visually. The algorithms have been taken from [6].

"h264ref" is the reference encoder implementation of the video standard H.264. H.264 is well-established standard for encoding videos. It performs macroblock-based motion estimation and -compensation. The "h264ref" ref-

Name	**Stages**	**Source**
"automotive"	21	*see Section 8.4*
"h264ref"	4	SPEC CPU 2006 [39]
"lame"	4	MiBench [38]
"PGP"	5	MiBench [38]
"Sphinx 3"	22	SPEC CPU 2006 [39]

Table 8.1: Overview of the benchmark applications which are used for the experiments.

erence implementation is part of the SPEC CPU 2006 benchmark suite [39] and has been parallelized into a software pipeline manually.

The audio encoder “lame”, which is part of the MiBench benchmark suite [38], is a free software encoder for converting uncompressed waveform data into MP3 audio. MP3 (MPEG-2 Audio Layer III) is a lossy audio encoding standard that uses a psychoacoustic model to discard information that is less likely to be noticed by humans. The application was parallelized manually to form a software pipeline which comprises 5 stages.

“PGP” is a data de-/encryption application (“Pretty Good Privacy”) based on the OpenPGP standard which has been published as RFC 4880 [29]. “PGP” performs data hashing and compression followed by symmetric-key and public-key cryptography. The implementation is part of the MiBench benchmark suite [38]. The parallel, software-pipelined version that was used for our experiments contains 5 stages.

“Sphinx 3”, which is part of the SPEC CPU 2006 benchmark suite [39], is a speech recognition application based on hidden markov models (HMM). We parallelized it manually to form a software pipeline that comprises 22 stages.

These applications have been selected because they perform complex computation on a continuous stream of input data. Furthermore, they represent a wide variety of applications from embedded object tracking over video-, audio-, and voice processing to data security. Additionally, the selected applications can be easily parallelized to form software pipelines.

8.5 Application Scenario

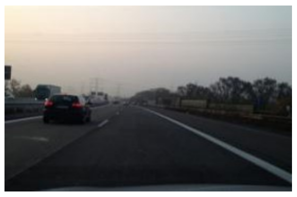
For the experiments, multiple instances of the benchmark applications are created so that the total number of stages exceeds the number of cores by at least a factor of 3 (this number was chosen arbitrarily to establish a considerable system load).

For systems that comprise 128, 512, and 1024 cores, this corresponds to 35, 138, and 275 concurrently running applications, respectively.

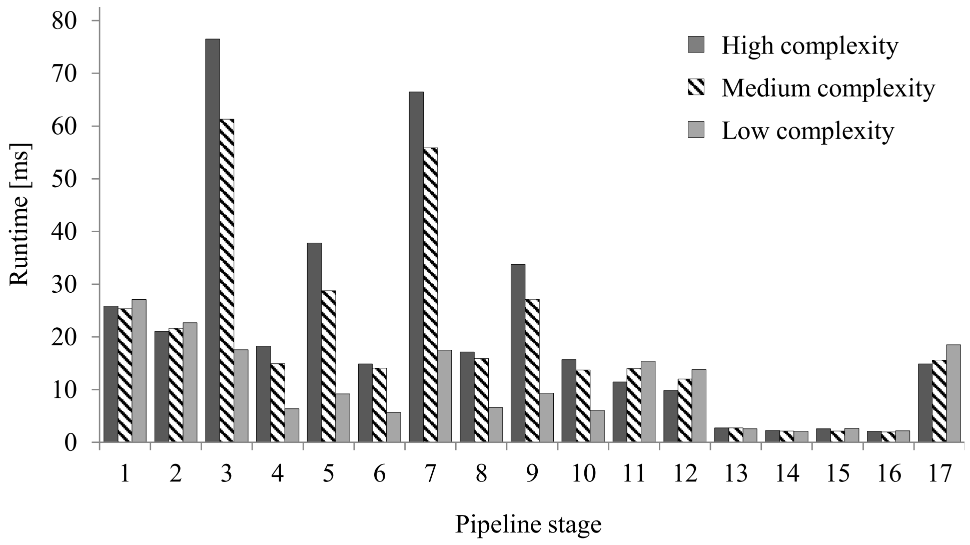
Each instance of a benchmark application is associated with a set of input data. For the automotive application, three different input video sequences of pre-captured color stereo video with a resolution of 640x480 pixels at 30 frames per second are used. The input video sequences correspond to high, medium, or low computational requirements of the application because the scene-inherent complexity differs largely among them. The impact of the different input scenes on the computational requirements of each stage is shown in Figure 8.3.

(a) High complexity

(b) Medium complexity



(c) Low complexity



(d) The computational requirements of the pipeline stages that result from the three different input scenes.

Figure 8.3: (a)-(c) show still images from three different input scenes (a color video sequence with a resolution of 640x480 in YUV format) for the "automotive" application. The computational complexity that is caused by these input scenes can be classified into high computational complexity (a), medium computational complexity (b), and into low computational complexity (c). The resulting computational requirements of the individual stages are shown in (d).

The input video sequences of the “h264ref” application are in the YUV format and include a popular reference video sequence, “Foreman” (QCIF resolution) and scenes from a movie published under the Creative Commons Attribution 3.0 license, “Big Buck Bunny” [91] in 1920x1080, 1280x720, and 854x840 resolution.

For “lame”, “PGP”, and “Sphinx 3”, the data that is included in the corresponding benchmark suites, the MiBench benchmark suite [38] and SPEC CPU 2006 [39], are used.

8.6 State-of-the-art Baselines

The resource allocation methods proposed in this thesis are compared to two state-of-the-art resource allocation methods, AIAC [7] and DistRM [64]. AIAC balances the computational load among cores by exchanging work items based on a distributed heuristic. DistRM shifts the responsibility to balance load among cores to the individual applications. It relies on the applications to supply a function that calculates their throughput for any given set of cores that it could be allocated. It then decides upon an allocation of cores to the applications, and the applications are responsible to allocate these cores to their individual tasks.

The rest of this section details how these methods for runtime resource allocation can be adapted to achieve a fair comparison with the methods proposed in this dissertation.

AIAC [7] This method for balancing computational load exchanges workload between physically neighboring cores to balance the computational load evenly. To adapt this method for software-pipelined applications in many core systems, workload is exchanged by migrating pipeline stages when the computational load is not balanced. This is achieved by comparing the load of adjacent cores and migrating a pipeline stage i when the difference of the summed computational requirements among all stages on each core exceeds c_i. To achieve a fair comparison, the assumption that only consecutive stages may be allocated to the same core is relaxed. For the adapted implementation of AIAC, a core may execute any stage from any application.

DistRM [64] This distributed resource management method distributes cores among applications, but relies on the applications to themselves decide how to distribute their tasks accordingly. Therefore, the optimal fusion algorithm proposed in Section 5.1 to achieve a fair comparison. Consequently, only the number of cores assigned to each application differs between DistRM and our methods, while the fusions of pipeline stages are carried out identically.

DistRM is adapted by using the tables according to Section 5.1. As DistRM remains in local optima if the speed-up of an application does not increase with another core (even if this was the case for a larger number of additional cores), marginal improvements are reported to the DistRM algorithm (an $\epsilon = 5 * 10^{-4}$ is chosen arbitrarily) as long as the number of cores does not exceed the number of stages of the corresponding application. Using the described adaptions, fair comparison to DistRM can be achieved.

8.7 Comparison to the State-of-the-art

In the following, the system throughput that results from using the methods for resource allocation that are presented in this thesis is compared to the state-of-the-art methods of AIAC [7] and DistRM [64]. This section focuses on two major scenarios: the first scenario allows to analyze how the resource allocation methods adapt to changing application scenarios. The second scenario allows to analyze the saturation of memory controllers when using the different methods, and the throughput that results from them.

Scenario: Adapting to unpredictable changes

In complex dynamic scenarios, applications may be started or stopped unpredictably and the resource requirements of their individual tasks may vary significantly. To compare the methods proposed in this thesis to the state-of-the-art methods AIAC [7] and DistRM [64], the resulting system throughput is compared for following scenario: during runtime, the input data for the applications is switched repeatedly, which results in changing resource requirements of the individual tasks. After 10 seconds, 9 out of 35 running applications (approx. 25%) are stopped. This scenario assumes a system with 128 cores where the memory controllers cannot be saturated. This is important to analyze the adaptivity of the methods in isolation, i.e. without

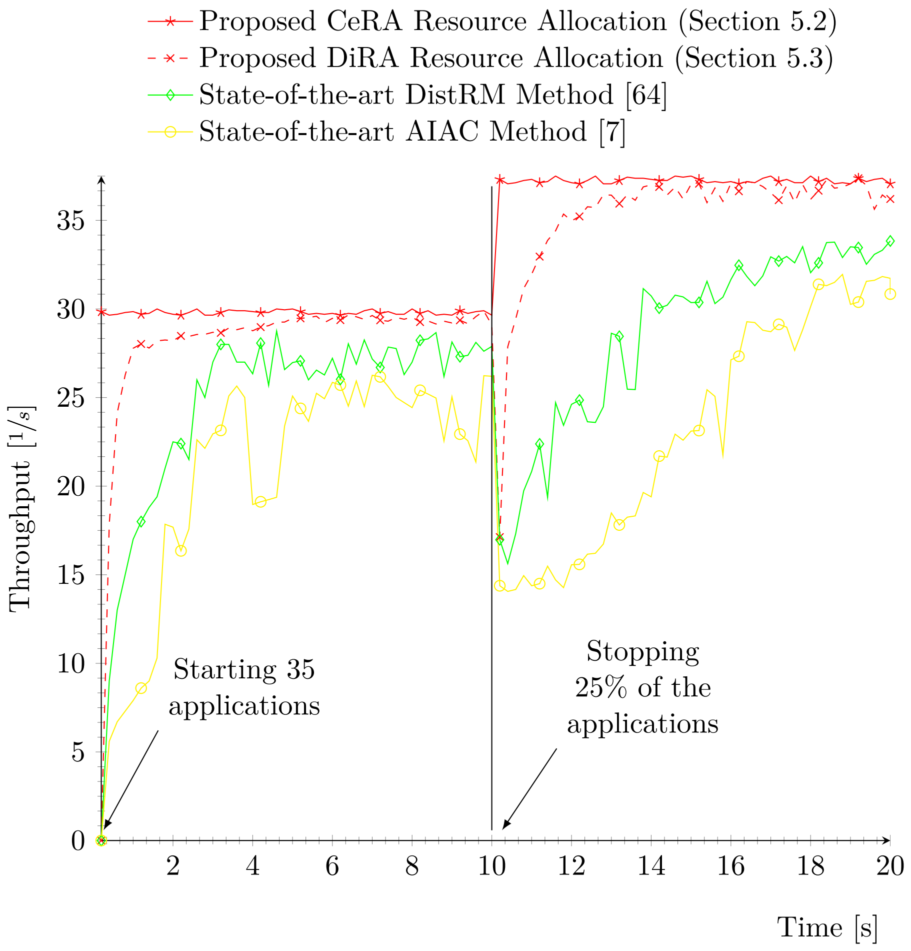


Figure 8.4: Comparison of the system throughput that results from using the resource allocation method proposed by this dissertation, AIAC [7], and DistRM [64]. For a scenario of a 128-core many-core system where the application scenario changes unpredictably after 10 seconds (25% of the 35 running applications are stopped), both the centralized method of Section 5.2 as well as the highly scalable, distributed method of Section 5.3 can achieve a significant improvement over the state-of-the-art methods AIAC [7] and DistRM [64].

effects from saturated memory controllers. Figure 8.4 shows the resulting system throughput when CeRA (Section 5.2) is used for allocating resources, when using DiRA (Section 5.3), AIAC [7], and when DistRM [64] allocate resources. As DiRA reduces the overhead by trading the optimum for a high degree of scalability, it results in a slightly reduced throughput as compared to the centralized method CeRA (Section 5.2).

Mainly due to its obliviousness to inter-task communication, AIAC [7] results in a much lower system throughput throughout the experiment. It does adapt the resource allocation when applications stop, but this adaption takes a considerable amount of time (approx. 3 seconds), and the maximum throughput amounts to only approx. 73% of the throughput that results when employing our centralized method. As DistRM [64] has been adapted in a way that fuses stages considering both computation and inter-task communication so that the throughput of an application is optimal given its set of cores, it significantly improves upon AIAC [7]. It results in a system throughput of approx. 82%-85% as compared to the centralized CeRA method, and approx. 91% compared to the distributed DiRA method.

Scenario: Comparing self-organization to system-controlled resource allocation

Self-organizing resource management uses local observations and actions to achieve global optimization without controlling instances. In the following, we compare our self-organizing Pipelets (Section 6.2) to our system-controlled CeRA method (Section 5.2) and to the system-controlled state-of-the-art DistRM [64] and AIAC [7] resource allocation methods. As AIAC [7] does not employ controlling instances and relies solely on local information, it can be regarded as a self-organizing method. For a system with 1024 cores, Figure 8.5 compares the throughput that results from using these methods. These results can be interpreted to quantify the performance loss of self-organizing approaches. It can be observed that the Pipelets significantly improve the throughput over the state-of-the-art AIAC [7] method by approx. 143%. In this experiment, the Pipelets result in a system throughput of approx. 63.1% of the throughput that results from the centralized CeRA method, which can be attributed to its lack of global information. Furthermore, the Pipelets result in a throughput of approx. 95.5% of the system-controlled, distributed DistRM [64] method. While they cannot achieve a system throughput that can be compared to methods that exploit global

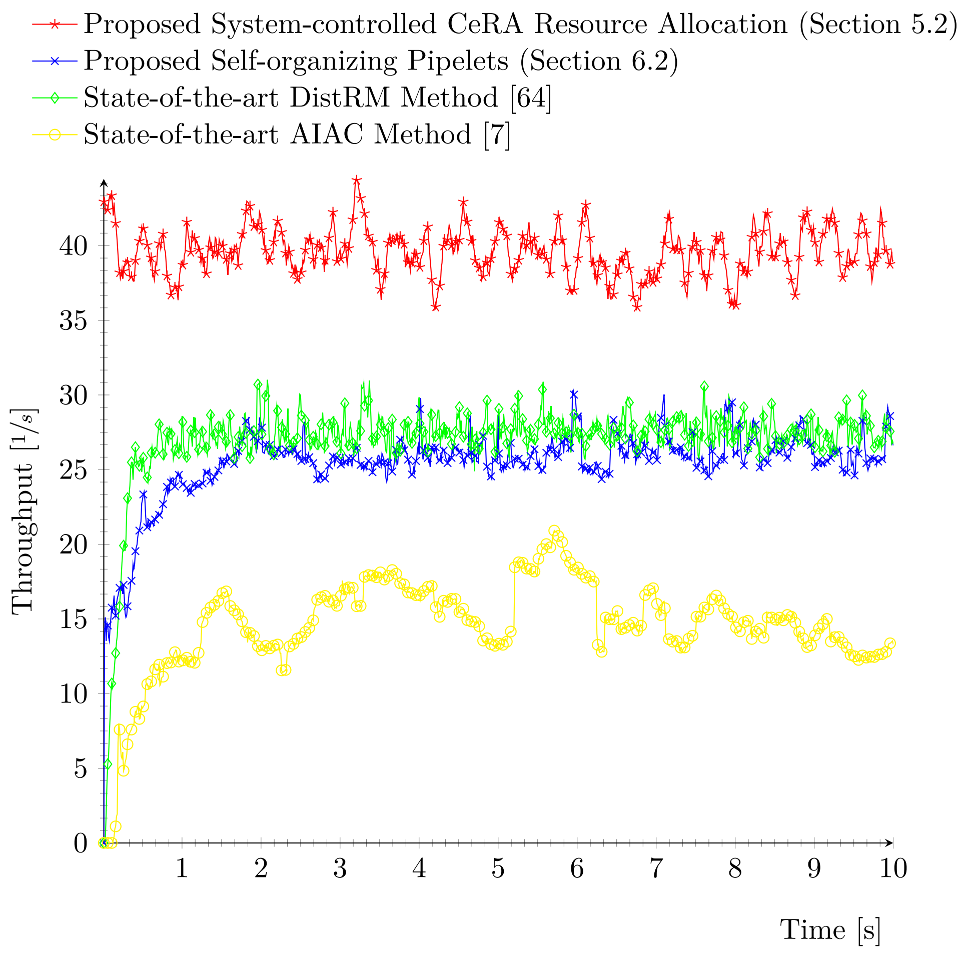


Figure 8.5: Comparison of self-organization to system-controlled resource allocation. The self-organizing Pipelets of Section 6.2 are compared to the centralized method of Section 5.2 as well as to the state-of-the-art resource allocation methods AIAC [7] and DistRM [64]. In this experiment of a system with 1024 cores, the Pipelets result in a throughput of approx. 64% of the centralized, system-controlled CeRA method. The distributed, system-controlled DistRM [64] method results in approx. 68% of the throughput of CeRA. The distributed AIAC [7] method results in a throughput of approx. 37% of CeRA.

information, Pipelets can be regarded as an effective method for allocating the resources of a many-core system in a self-organizing way.

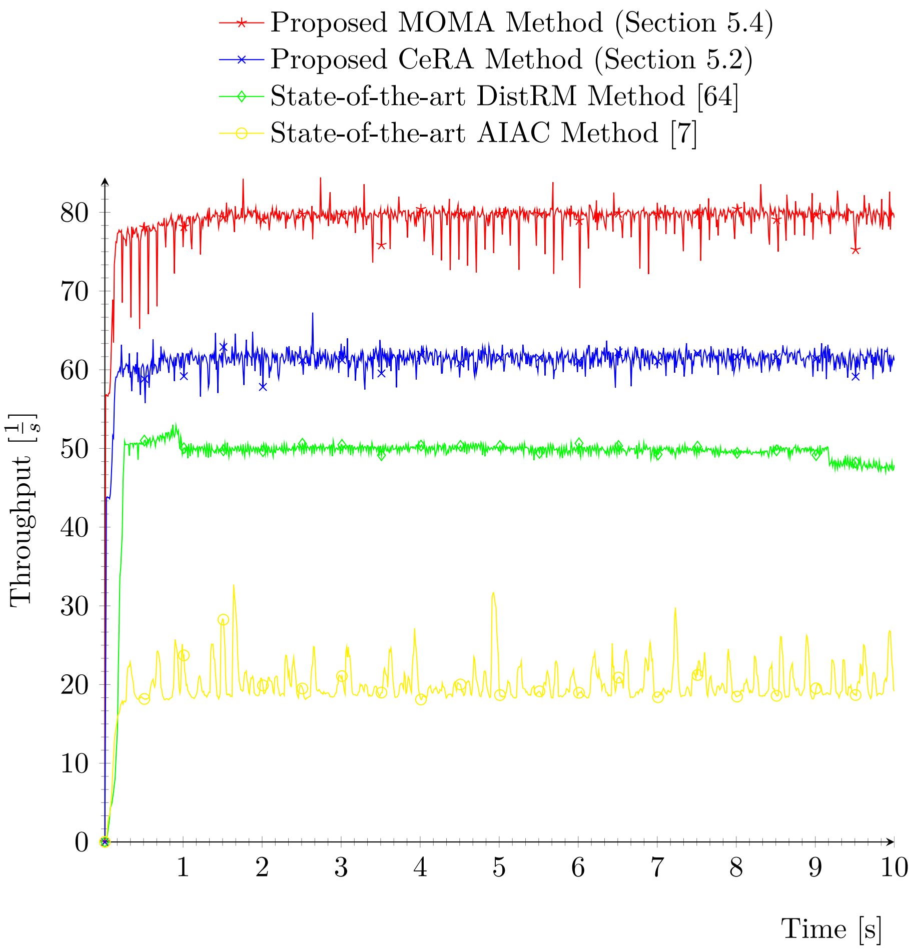


Figure 8.6: Comparison of the system throughput when balancing the load of the memory controllers in a system with 1024 cores. In this experiment, each memory island contains 64 cores. MOMA improves the throughput by 1.29x over CeRA, by 1.61x over DistRM [64], and by 3.95x over AIAC [7].

Scenario: Saturated memory controllers

When multiple memory-intensive applications run concurrently, memory controllers may operate in saturation. This may occur especially in systems that comprise a large number of cores and thus, when the cores that belong to each memory island may require more memory bandwidth than the memory controller can deliver. In the following, the MOMA method proposed in Section 5.4 to jointly optimize for computation, communication and memory bandwidth is compared to CeRA, DeRA and to the state-of-the-art methods AIAC [7] and DistRM [64]. This experiment is performed for systems with 1024, 512, and 128 cores.

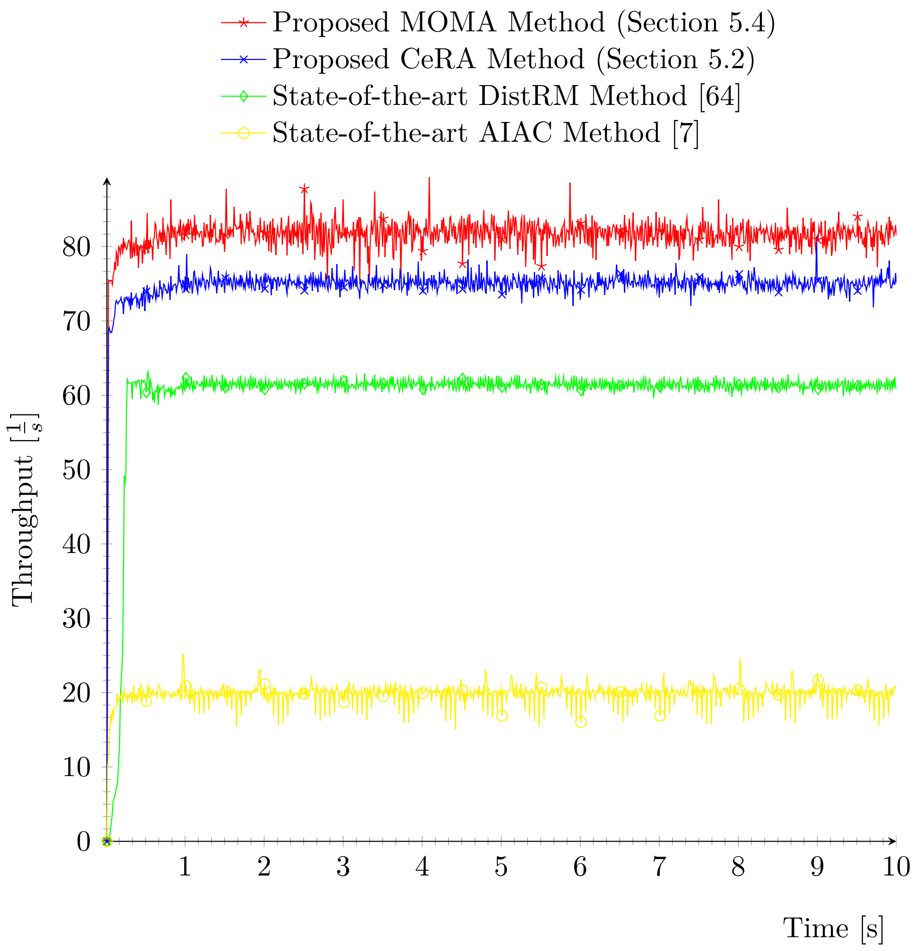


Figure 8.7: Comparison of the system throughput when balancing the load of the memory controllers in a system with 512 cores. In this experiment, each memory island contains 32 cores. MOMA improves the throughput by 1.09x over CeRA, by 1.35x over DistRM [64], and by 4.11x over AIAC [7].

Figure 8.6 compares the system throughput of the resource allocation methods proposed in this thesis to the state-of-the-art for a system with 1024 cores and 16 memory islands. Each memory islands contains 64 cores and when resource allocation does not take the bandwidth capacity of memory controllers into account, memory controllers may operate in saturation. This can severely degrade the throughput of the system. In this experiment, the heuristic proposed in Section 5.4 improves the system throughput by 1.29x over the CeRA resource allocation of Section 5.2, by 1.61x over DistRM [64] and by 3.95x over AIAC [7], respectively. Thus, a significant improvement can be observed.

Figure 8.7 shows the system throughput that can be observed in an experiment for a system that comprises 512 cores and 16 memory islands (each memory island contains 32 cores). The improvement in system throughput that results from balancing the memory requirements of the individual stages among the memory islands amounts to 1.09x as compared to the CeRA resource allocation method of Section 5.2, 1.35x as compared to DistRM [64], and 4.11x as compared to AIAC [7].

For a system that comprises 128 cores, however, the 8 cores that form a memory island cannot force their memory controller to operate in saturation. As a result, heuristically balancing the memory requirements of the individual stages among the memory islands results in a degradation of approx. 5% over the CeRA resource allocation method of Section 5.2. An improvement of 2.13x and 1.12x can be observed over AIAC [7] and DistRM [64], respectively.

Figures 8.9, 8.10, and 8.11 show the average variance of the load of all memory controllers over time, which serves as an indicator for their load balance, in order to explain the improved system throughput that can be observed. In a system that comprises 1024 cores, the variance of the load of memory controllers is significantly reduced as compared to the CeRA resource allocation method of Section 5.2 and DistRM [64]. The average variance of the load of the memory that results from employing AIAC [7] is lower because AIAC [7] results in a low throughput of the applications and thus, their memory requirement is lower.

As shown in Figure 8.10, the gap between the variances is smaller in systems that comprise 512 cores. In this experiment, the memory-aware resource allocation of Section 5.4 and the CeRA resource allocation method of Section 5.2 achieve a very low variance and thus a good load balance among the memory controllers.

8.8 Computational Overhead

This section assesses the computational overhead of the methods proposed in this thesis.

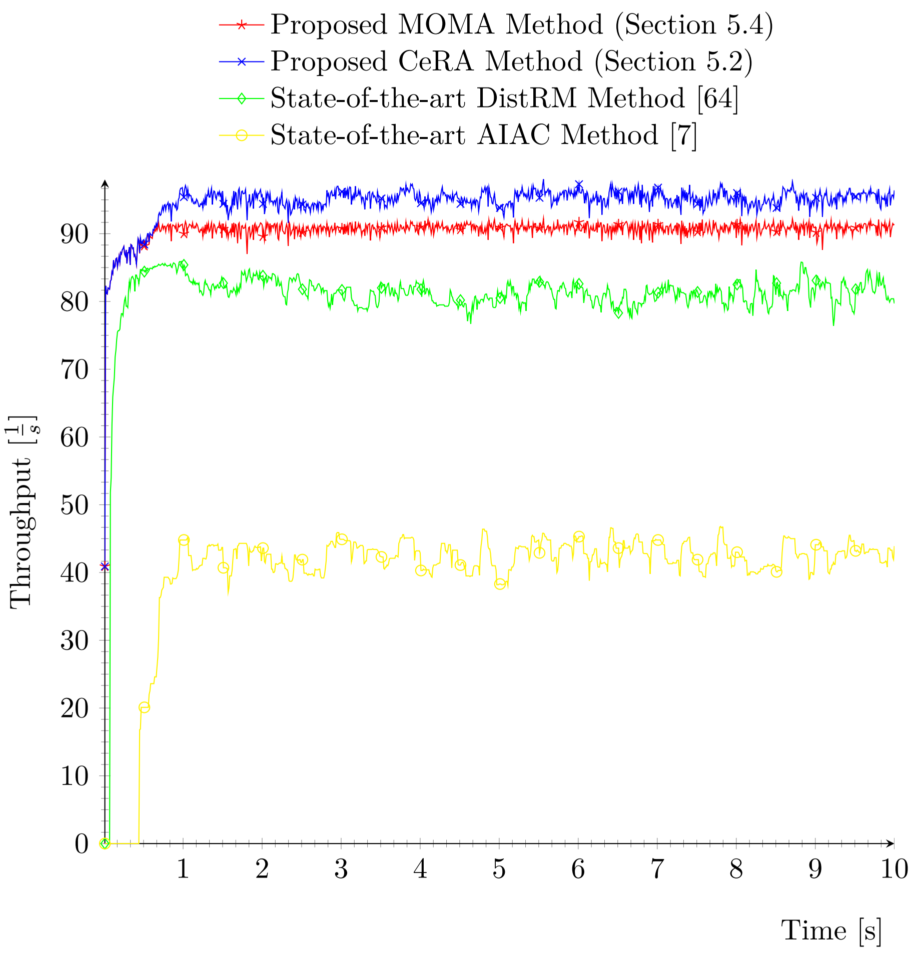


Figure 8.8: Comparison of the system throughput when balancing the load of the memory controllers in a system with 128 cores. In this experiment, each memory island contains 16 cores. As the memory controllers never operate in saturation, MOMA decreases the system throughput by 0.95x over CeRA. Over DistRM [64] and AIAC [7], MOMA improves the throughput by 1.12x and 2.13x, respectively.

Figure 8.12 compares the computational overhead of the CeRA resource allocation method of Section 5.2, to the computational overhead of the distributed DiRA resource allocation method of Section 5.3. The computational overhead of CeRA grows to approx. 37 seconds for allocating the resources of a system with 1024 cores and 128 applications. This high computational requirement results from the computational complexity of $O(K, M^2)$ where K is the number of applications and M is the number of cores. Table 8.2 contains measured values for this experiment. Infeasible combinations, i.e. combinations where the number of applications exceeds

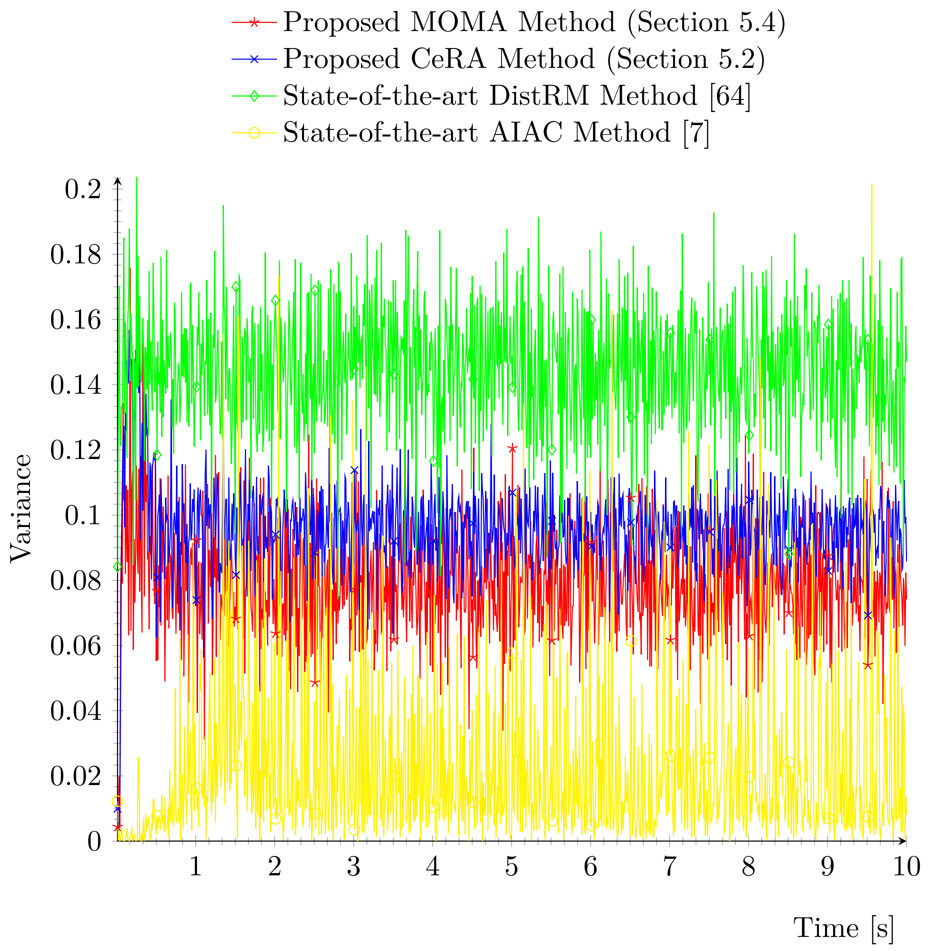


Figure 8.9: Comparison of the variance between the load of the memory controllers in a system with 1024 cores.

the number of cores, which violates the assumption of $K \leq M$, are in parentheses.

8.9 Communication Overhead

The communication overhead of using the proposed CeRA (Section 5.2) and DiRA (Section 5.3) methods is shown in Figure 8.13. The proposed MOMA

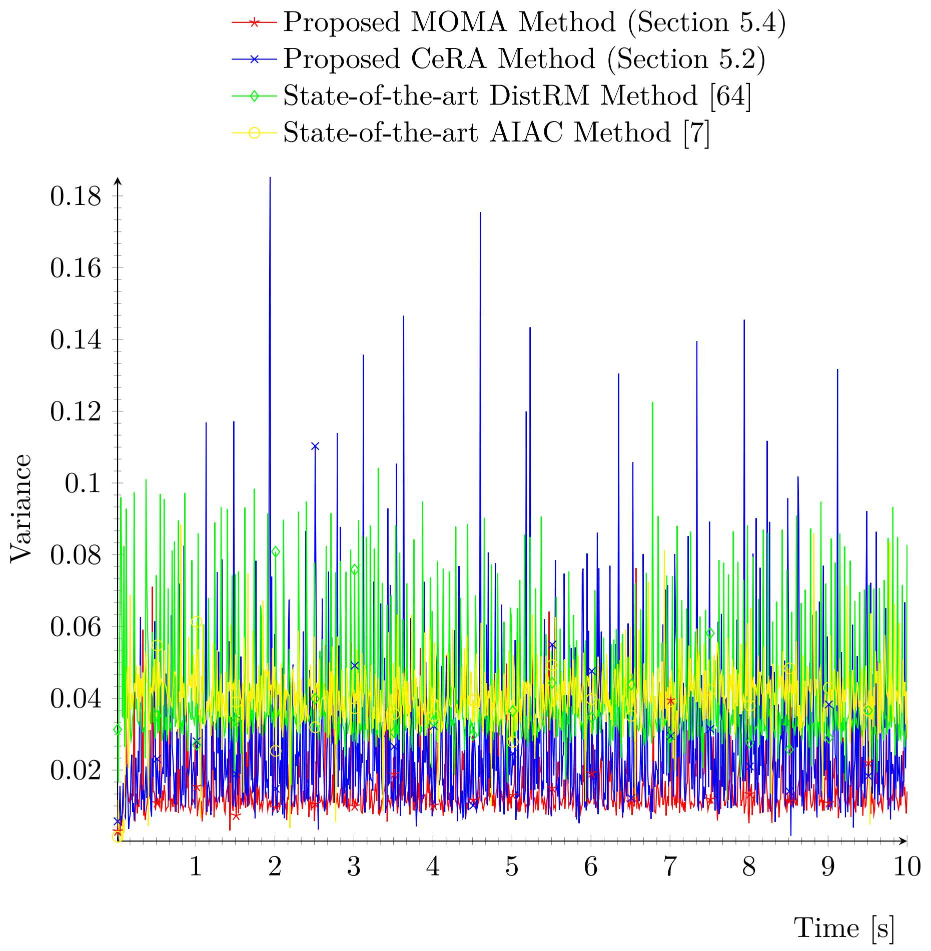


Figure 8.10: Comparison of the variance between the load of the memory controllers in a system with 512 cores.

# Cores	**# Applications**					
	4	**8**	**16**	**32**	**64**	**128**
4	0.00	-	-	-	-	-
8	0.01	0.01	-	-	-	-
16	0.01	0.02	0.05	-	-	-
32	0.01	0.03	0.07	0.15	-	-
64	0.02	0.05	0.12	0.27	0.49	-
128	0.05	0.11	0.22	0.51	1.00	1.92
256	0.11	0.25	0.56	1.12	2.19	4.14
512	0.29	0.69	1.39	2.81	5.73	11.33
1024	0.93	1.88	4.32	9.21	17.89	37.32

(a) Runtime of the proposed CeRA method (Section 5.2) [s]

# Cores	Runtime
16	0.18
32	0.30
64	0.55
128	0.44
256	0.64
512	0.57
1024	0.82

(b) Runtime of the proposed DiRA method (Section 5.3) [ms] for all numbers of applications

Table 8.2: The measured computational overhead of the proposed CeRA method of Section 5.2 in seconds and of the distributed, hierarchical DiRA method of Section 5.3 in milliseconds. The runtime of the DiRA method does not depend on the number of applications. Infeasible combinations (i.e. more applications than cores) are denoted by a dash.

method to account for the bandwidth capacity of memory controllers (Section 5.4) does not introduce additional overhead and thus, its overhead is identical to the overhead of CeRA.

The communication overhead of CeRA and DiRA reach approx. 0.25‰ and approx. 0.1‰, respectively, of the total communication volume (i.e. all communication that is observed in the system, which includes the overhead of the resource allocation methods as well as the inter-core communication of the running applications). This corresponds to approx. 365 KB/s and 138

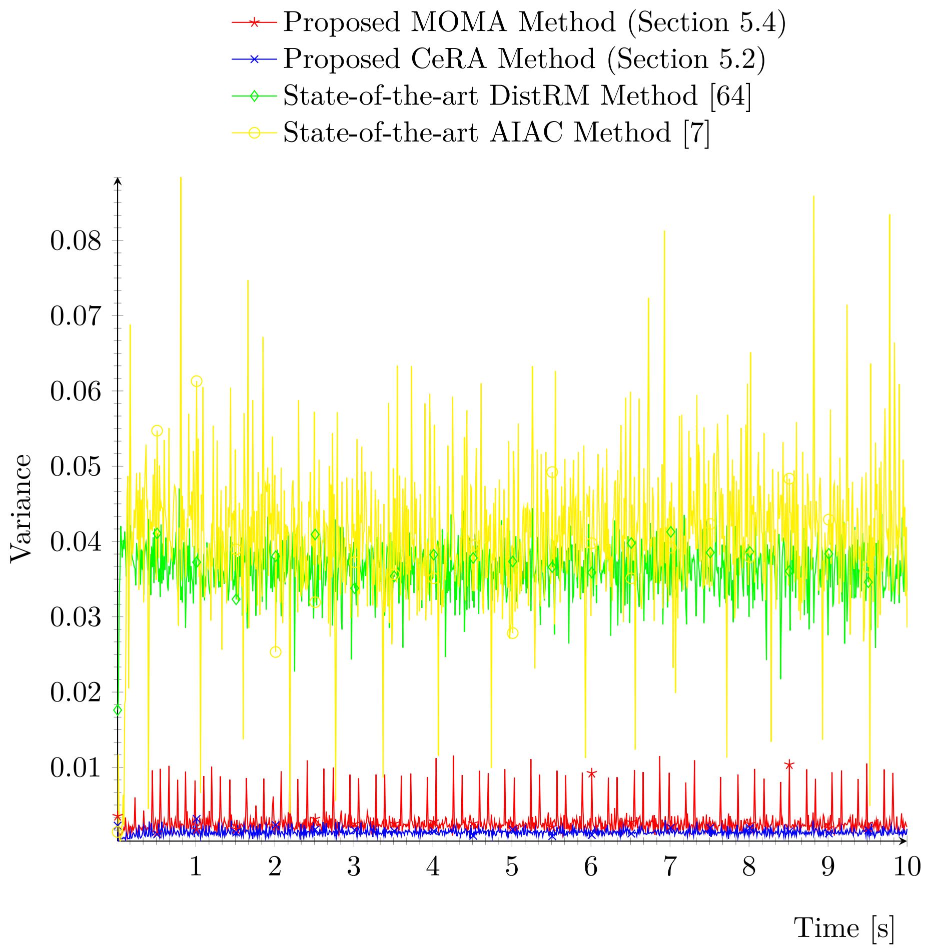


Figure 8.11: Comparison of the variance between the load of the memory controllers in a system with 128 cores.

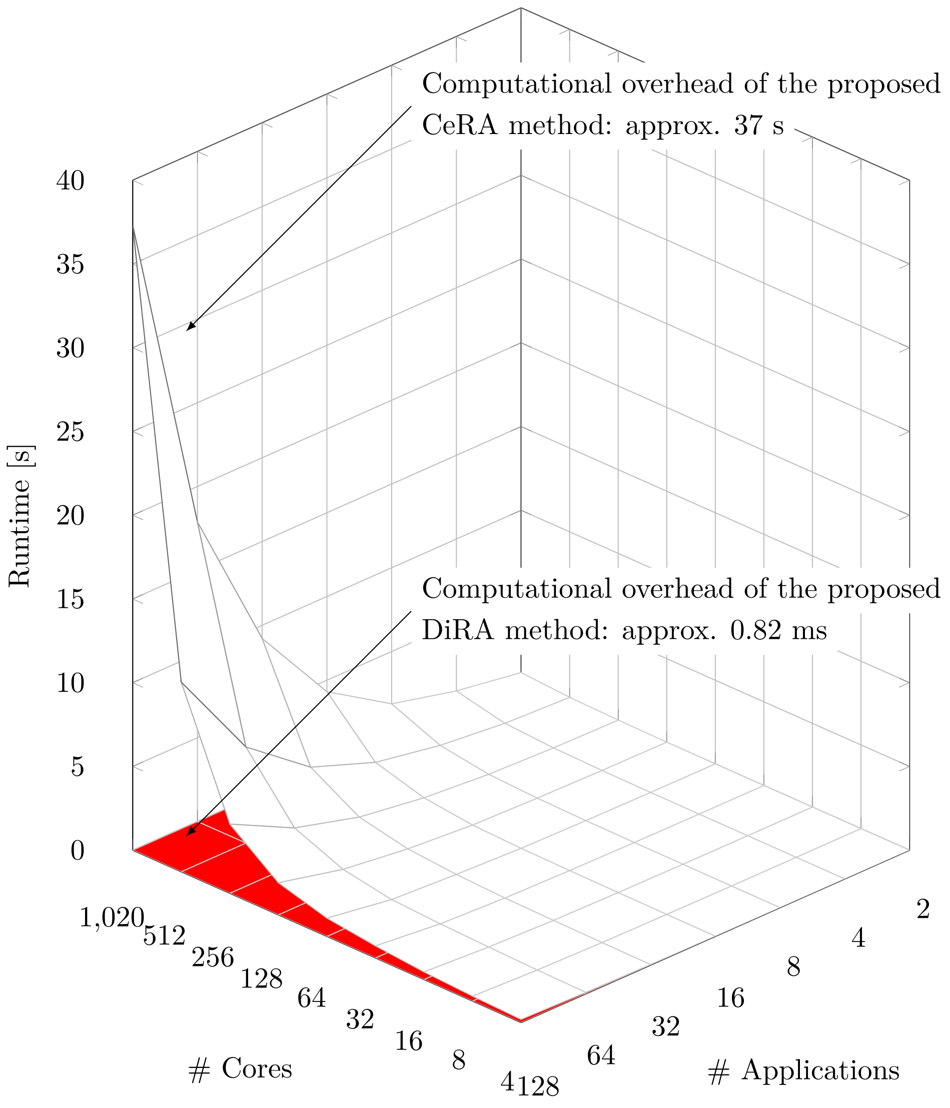


Figure 8.12: Comparison of the computational overhead that results from the centralized CeRA resource allocation method of Section 5.2 to the computational overhead of the distributed, hierarchical DiRA resource allocation method of Section 5.3.

KB/s of communication volume for the methods, respectively, as compared to the total communication volume of approx. 1455 MB/s for a system that comprises 1024 cores and 275 concurrently running applications with a total of 3080 stages.

8.10 Task Migration

The CARAT task migration mechanism proposed in Chapter 7 forms a foundation for adapting resource allocations at runtime. The measured overheads for adapting resource allocations, which corresponds to the overhead of fusions and fissions of stages, are shown in Table 8.4.

The column "Carried State" contains the size of data, in KB, that must be transferred between cores when a task is migrated. This includes the size of the stack and registers and the heap memory that is transferred by CARAT, but excludes the size of the executable file. The transfer of the executable

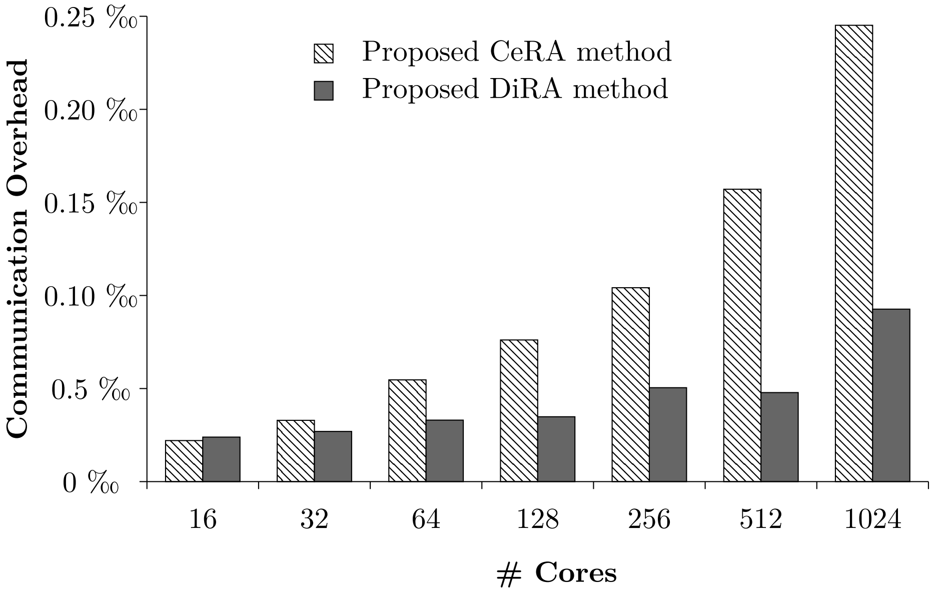


Figure 8.13: A comparison of the communication overhead of the proposed CeRA (Section 5.2) and DiRA (Section 5.4) methods for resource allocation, expressed as a percentage of the total communication volume of the system. Table 8.3 contains the measured values.

# Cores	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Applications	5	10	20	35	70	140	275
Stages	56	112	224	392	784	1568	3080
App. Comm [MB/s]	28.3	56.6	98.9	198.1	367.3	792.4	1455.0
CeRA (Section 5.2) [KB/s]	0.64	2.27	5.53	15.2	39.2	178.5	365.0
DiRA (Section 5.4) [KB/s]	0.69	1.58	3.34	6.79	19.0	54.2	138.0

Table 8.3: Communication overhead [KB/s] of our CeRA (Section 5.2) and DiRA (Section 5.4) methods for resource allocation and the total communication volume of the running applications [MB/s] for systems with 16, 32, 64, 128, 256, 512 and 1024 cores.

file is excluded because in many cases, this executable is already loaded on the destination core.

This is the case for fusion operations and for fissions in case any stage of the application is already allocated to the destination core. This is expressed by the columns “Old Core” and “New Core”, where “New Core” means that the executable file must be loaded and started, and “Old Core” expresses otherwise.

8.11 Summary of Experiments and Evaluations

In this chapter, the experiments that have been conducted to compare the contributions of this thesis to the state of the art have been illustrated and discussed. The experimental evidence shows that the proposed methods can significantly improve upon existing methods in many scenarios. Furthermore, they show that this can be achieved with a very low runtime overhead, both in terms of computation as well as in terms of communication. Complex, real-world applications have been used for all evaluations so that reliable evidence was obtained.

	Carried State [KB]				**Overhead [ms]**	
Application	**Min**	**Max**	**Avg**	σ	**Old Core**	**New Core**
"automotive"	1	32	19	15.21	0.63	22
"h264ref" [39]	13	53	27	22.73	1.07	76
"lame" [38]	9	10	9	1.32	0.18	19
"PGP" [38]	1	27	12	9.11	0.30	66
"Sphinx 3" [39]	12	22	17	4.21	0.51	44

Table 8.4: Size of the state, in KB, that is carried among processing multiple data items and the resulting overhead of fusing stages. When a core is allocated to the application, it may have been allocated to the application before ("Old Core") or not ("New Core"). In the latter case, the overhead is significantly larger as the executable file needs to be started on this core.

8.12 Critical Discussion and Limitations

Main Advantages The methods and algorithms proposed in this thesis compose a system that allows to effectively and efficiently allocate the resources of many-core systems. It addresses complex software-pipelined, memory-intensive applications even in dynamic scenarios. As a result, this system leads to optimal or near-optimal throughput in many cases. Significant improvements over the state-of-the-art can be observed as the methods and algorithms proposed in this thesis optimize jointly for computation and communication while accounting for the limited bandwidth of multiple memory controllers. A specific focus is put on distributed, robust resource allocation in a way that is resilient to malfunctioning cores.

Furthermore, this thesis shows how resource allocation can adapt to drastic changes in the resource requirements of applications at runtime. This adaptivity addresses scenarios where applications may be started or stopped and may change their resource requirements drastically and unpredictably at any time. This thesis shows how complex scenarios that consist of multiple parallel, complex, memory-intensive applications can be addressed. These advantages have been extensively described, argued and verified experimentally in this thesis.

Limitations Despite these important advantages, the methods and algorithms proposed in this thesis face some limitations in their applicability. Firstly, this thesis assumes that all applications form software pipelines. A linear execution model is assumed where each stage has at most one predecessor and one successor. This limitation is mainly induced by the application model of Section 4.1. This model allows to calculate the throughput of each application for all possible fusions of stages. In order to alleviate this limitation, future work should focus on extending this model. Ultimately, when applications could calculate their throughput and the allocation of their tasks to a given number of cores, the applicability of the methods and algorithms proposed in this thesis could be greatly extended for a broader spectrum of applications beyond software pipelines.

Another limitation is posed by the focus on the performance of a system. This thesis regards performance as a central optimization criterion and does not address concerns such as thermal management or power consumption. While performance is often a key factor, both concerns are of considerable importance in some systems. In such cases, this thesis relies on other, orthogonal means to address these concerns, e.g. Dynamic Voltage and Frequency Scaling (DVFS) and sufficient external cooling.

Furthermore, the algorithms and methods proposed in this thesis inherently operate on a *temporal granularity*, i.e. the minimum time between adapting resource allocations. As such, the algorithms observe the system state and potentially adapt resource allocations each time a data item has been processed by a stage. Hence, this implicitly assumes a considerable throughput. In scenarios where all running applications have very low throughputs, resource allocation can only be adapted infrequently.

Lessons Learned From a careful examination of the advantages and limitations of the methods and algorithms proposed in this thesis, a number of lessons can be learned. Most importantly, this thesis shows that resource allocation plays a key role for the performance of many-core systems. As such, it is of crucial importance for the success of many systems, including but not limited to embedded many-core systems.

The role of resource allocation becomes even more important when future systems integrate a growing number of cores, when applications become more and more complex and memory-intensive, and when hardware becomes unreliable.

Furthermore, this thesis strongly supports the finding that the computational requirements of tasks, their communication behavior, as well as their bandwidth requirements need to be accounted for jointly.

Despite the contributions of this thesis, important challenges remain for applications that cannot be modeled efficiently as software pipelines, and for systems where the performance is not the most important objective.

Chapter 9

Conclusion

This thesis presents novel concepts, methods, and mechanisms for efficiently allocating the resources of many-core systems. The contribution of this thesis improves upon the state of the art by exploiting properties of software pipelines for resource allocation.

Specifically, the centralized CeRA method and its distributed, hierarchical extension DiRA are proposed to allocate resources. Recent research along with this thesis identifies a possible saturation of memory controllers as a major issue for the performance of large many-core systems. Hence, this thesis addresses this challenge by proposing the MOMA method. MOMA allows to optimize resource allocation jointly for computation, communication, and for avoiding the saturation of memory controllers in systems with multiple memory islands.

Additionally to these system-controlled methods, self-organizing methods for resource allocation are proposed. Self-organization is a powerful paradigm to manage complex systems. To exploit the benefits of self-organization, this thesis proposes Pipelets as self-organizing stages of software pipelines. Pipelets allocate system resources at runtime through local observations and limited interactions. As a next step, self-organizing software pipelines are proposed to combine the benefits of system-controlled and self-organizing methods. They form an extension to the DiRA method and employ self-organization to increase the resilience of resource allocation against unreliable hardware. This way, ad-hoc interaction which is limited to spatial and temporal proximity can alleviate the problems that arise from unreliable cores, while experiments substantiate near-optimal performance.

In order to accurately model and to efficiently estimate the throughput of a system for a given resource allocation, the contributions of this thesis exploit properties of software pipelines. The most important properties that are exploited are the linear execution pattern of software pipelines as well as the fact that the slowest stage of a pipeline limits its throughput.

The proposed methods are able to respond to changing resource requirements of tasks and to changing application scenarios by adapting resource allocations at runtime. Such an adaption of resource allocation necessitates efficient task migration. To address this issue, a novel mechanism for task migration, CARAT, is presented that allows to frequently adapt resource allocations at runtime.

The experiments show that the proposed methods allow to significantly increase the performance of a many-core system over the state of the art. Furthermore, very low overhead both in terms of computation and communication can be observed, which strongly supports the formal complexity analysis of the proposed algorithms. Hence, efficient resource allocation can be performed even for systems with more than thousand cores.

This dissertation allows to draw the following conclusions:

- Resource allocation can benefit greatly from exploiting properties of software pipelines.
- It is crucial to jointly optimize for computation, communication, as well as for the saturation of memory controllers in order to efficiently employ many-core systems.
- For large systems, optimal resource allocations are hardly possible. However, near-optimal resource allocations can be achieved with low overheads.

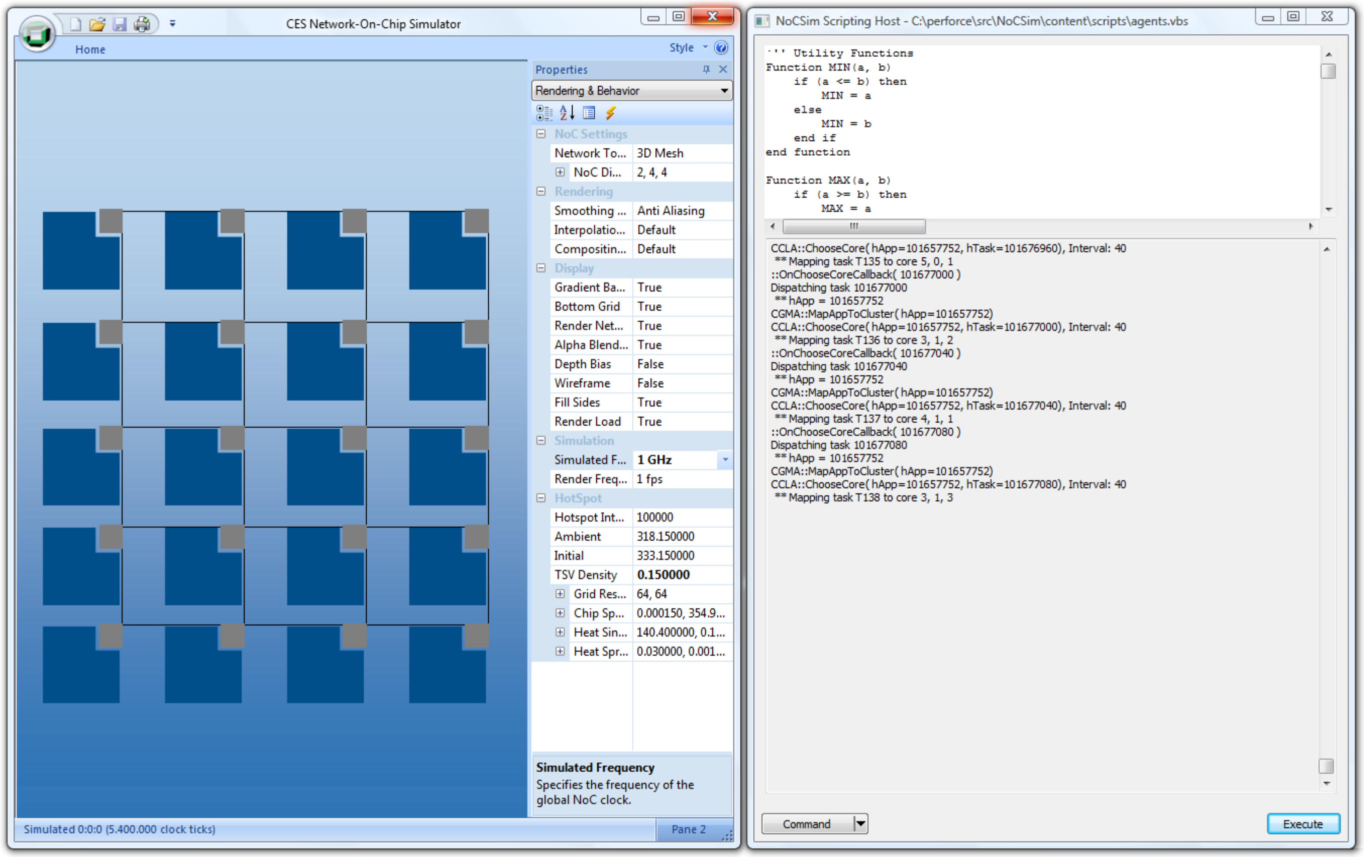
Appendix A

Many-core System Simulator

In this thesis, a behavioral high-level many-core system simulator is used for many experiments. It is written in C++ and executes application traces that contain traces of computation, communication, and memory accesses.

The computation and communication traces have been obtained on the Intel Single-Chip Cloud Computer, while memory traces have been obtained as detailed in Appendix B. It simulates Intel P54C cores that are interconnected with a 2D mesh Network-on-Chip (NoC). The parameters for the link width (16 Bytes), router latency (2 hops) and frequency (1.6 GHz) correspond to the properties of the Intel Single-Chip Cloud Computer. Furthermore, it simulates 16 equally-sized memory islands, where each memory controller has a bandwidth constraint of 128 GiB/s, similar to the bandwidth constraint of many state-of-the-art GDDR5 controllers. Similar to Intel's Single-Chip Cloud Computer [43], the comparably slow individual cores cannot saturate the fast memory controllers in systems with 128 cores and less (8 cores per memory island) [79].

Inside this simulator, the proposed resource allocation methods CeRA (Section 5.2), DiRA (Section 5.3), MOMA (Section 5.4), as well as the state-of-the-art methods DistRM [64] and AIAC [7] have been implemented. To migrate tasks between cores, data with the size of the task context is sent over the NoC. Each application periodically reports its throughput.


CES Network-On-Chip Simulator
Home
Style
Properties
Rendering & Behavior
Simulated 0:0:0 (5.400.000 clock ticks)
Pane 2
NoCSim Scripting Host - C:\perforce\src\NoCSim\content\scripts\agents.vbs
Command
Execute

NoC Settings	
Network To...	3D Mesh
NoC Di...	2, 4, 4
Rendering	
Smoothing ...	Anti Aliasing
Interpolatio...	Default
Compositin...	Default
Display	
Gradient Ba...	True
Bottom Grid	True
Render Net...	True
Alpha Blend...	True
Depth Bias	False
Wireframe	False
Fill Sides	True
Render Load	True
Simulation	
Simulated F...	1 GHz
Render Freq...	1 fps
HotSpot	
Hotspot Int...	100000
Ambient	318.150000
Initial	333.150000
TSV Density	0.150000
Grid Res...	64, 64
Chip Sp...	0.000150, 354.9...
Heat Sin...	140.400000, 0.1...
Heat Spr...	0.030000, 0.001...

Simulated Frequency
Specifies the frequency of the global NoC clock.

```
''' Utility Functions
Function MIN(a, b)
    if (a <= b) then
        MIN = a
    else
        MIN = b
    end if
end function

Function MAX(a, b)
    if (a >= b) then
        MAX = a
```

```
CCLA::ChooseCore( hApp=101657752, hTask=101676960), Interval: 40
 ** Mapping task T135 to core 5, 0, 1
::OnChooseCoreCallback( 101677000 )
Dispatching task 101677000
 ** hApp = 101657752
CGMA::MapAppToCluster( hApp=101657752)
CCLA::ChooseCore( hApp=101657752, hTask=101677000), Interval: 40
 ** Mapping task T136 to core 3, 1, 2
::OnChooseCoreCallback( 101677040 )
Dispatching task 101677040
 ** hApp = 101657752
CGMA::MapAppToCluster( hApp=101657752)
CCLA::ChooseCore( hApp=101657752, hTask=101677040), Interval: 40
 ** Mapping task T137 to core 4, 1, 1
::OnChooseCoreCallback( 101677080 )
Dispatching task 101677080
 ** hApp = 101657752
CGMA::MapAppToCluster( hApp=101657752)
CCLA::ChooseCore( hApp=101657752, hTask=101677080), Interval: 40
 ** Mapping task T138 to core 3, 1, 3
```

Figure A.1: Screenshot of the high-level many-core system simulator

Appendix B

Sampling of Memory Accesses

B.1 Trace File Formats

B.1.1 Memory Traces

Memory traces are binary files. The data structure used for storing the entries is shown in Listing B.1: The first field, `cType`, contains a `1` if the entry corresponds to an allocation, a `2` for a free, a `3` for a read and a `4` for a write access. The field `cThreadID` contains the ID of the thread that allocated/freed/accessed the memory. This is relevant for multi-threaded programs only. The `uTimestamp` is a 64-bit unsigned integer that contains the number of clock cycles elapsed since starting the application. `pAddr` contains the address of the memory that is allocated/freed/accessed, and `pCodeAddr` contains the address of the corresponding executable instruction. `pCoreAddr` can be used to find the position in the source code that is responsible for the memory access or allocation/free. The last field of the structure is a union that contains either the base address (`pBaseAddr`) of the accessed memory block, or its size (`uSize`), depending on the value of `cType`. For `cType` $\in \{1, 2\}$, `uSize` is valid. Otherwise, `pBaseAddr` is valid.

Using the file format described above, accurate memory traces can be stored easily. The constant size of the memory structure allows to pre-allocate a buffer without requiring dynamic memory allocation at runtime, which significantly simplifies the collection of memory traces.

Listing B.1 The data structure used for storing memory traces

```
typedef struct {
  unsigned char cType;       // access type
  unsigned char cThreadID;   // thread ID
  unsigned __int64 uTimestamp; // time stamp
  void *pAddr;               // memory location
  void *pCodeAddr;           // code address
  union {
    void *pBaseAddr;         // base address
    unsigned int uSize;      // size of the block
  };
} MEMLOG, *LPMEMLOG;
```

B.1.2 Execution Traces

Execution traces of applications are stored in text files. Each file contains a header, followed by a collection of stage information structures. The structure of the file header is shown in Listing B.2: the number of stages of the software-pipelined application is denoted by `NUM_STAGES`, the `PROGRAM_PATH` contains the absolute path of the executable. Furthermore, `COMMAND_LINE` contains the command line used for collecting the execution trace. `DATE` contains the ISO 8601-formatted date string of the collection.

Listing B.2 Execution trace file header

```
NUM_STAGES = integer   // number of stages
PROGRAM_PATH = string  // executable path
COMMAND_LINE = string  // command line
DATE = string          // date [ISO 8601]
```

Similarly to the file header, each stage is described by an information structure shown in Listing B.3. Each stage information structure starts with a string, `[STAGE]`, followed by fields that characterize the stage: `INPUT_SIZE` contains the number of bytes that are received from the direct predecessor, if applicable, for processing a data item. `COMPUTE_TIME` contains the number of milliseconds required for the computation of the stage. `OUTPUT_SIZE` contains the number of bytes that are sent to the direct successor, if applicable, for each data item that has been processed. In order to allow a co-simulation of execution and memory traces, `MEM_TRACE_FILE` contains the path to the corresponding memory trace.

Listing B.3 Execution trace stage structure

```
[STAGE]
INPUT_SIZE = integer    // input size [bytes]
COMPUTE_TIME = double   // time [ms]
OUTPUT_SIZE = double    // output size [bytes]
MEM_TRACE_FILE = string // path to memory trace
```

Bibliography

[1] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving Predictable Performance through Better Memory Controller Placement in Many-Core CMPs. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 451–461, 2009.

[2] Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications. *EURASIP Journal on Embedded Systems*, 2008:9:1–9:15, 2008.

[3] Lluc Alvarez, Lluís Vilanova, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. Hardware-software Coherence Protocol for the Coexistence of Caches and Local Memories. In *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 89:1–89:11, 2012.

[4] Yeshayahu Artsy and Raphael Finkel. Designing a Process Migration Facility: The Charlotte Experience. *Computer*, 22(9):47–56, 1989.

[5] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A Compiler and Runtime for Heterogeneous Computing. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 271–276, 2012.

[6] Pedram Azad, Tilo Gockel, and Rüdiger Dillmann. *Computer Vision - Principles and Practice.* Elektor Electronics, 2008.

[7] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. Dynamic Load Balancing and Efficient Load Estimators for Asyn-

chronous Iterative Algorithms. *IEEE Transactions Parallel Distributed Systems*, 16:289–299, April 2005.

[8] Michael A. Baker and Karam S. Chatha. A Lightweight Run-time Scheduler for Multitasking Multicore Stream Applications. In *IEEE International Conference on Computer Design (ICCD)*, pages 297–304, 2010.

[9] Mohamed A. Bamakharma and Todor Stefanov. Managing Latency in Embedded Streaming Applications under Hard-Real-Time Scheduling. In *Proceedings of the ACM International Symposium on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, pages 83–92, 2012.

[10] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable Cluster Computing with MOSIX for Linux. In *Proceedings of the Linux Expo*, pages 95–100, 1999.

[11] Davide B. Bartolini, Riccardo Cattaneo, Gianluca C. Durelli, Martina Maggio, Marco D. Santambrogio, and Filippo Sironi. The Autonomic Operating System Research Project: Achievements and Future Directions. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, 2013.

[12] Luis Angel Bathen and Nikil Dutt. HaVOC: A Hybrid Memory-aware Virtualization Layer for On-chip Distributed ScratchPad and Non-volatile Memories. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 447–452, 2012.

[13] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages 1–6, 2006.

[14] Alessio Bonfietti, Luca Benini, Michele Lombardi, and Michela Milano. An Efficient and Complete Approach for Throughput-maximal SDF Allocation and Scheduling on Multi-Core Platforms. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages 897–902, 2010.

[15] S. Borkar. Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation. *Proceedings of the IEEE International Symposium on Microarchitecture (MICRO)*, 25(6):10 – 16, nov.-dec. 2005.

[16] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999.

[17] Shekhar Borkar. Thousand Core Chips: a Technology Perspective. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 746–749, 2007.

[18] D. Brooks, R.P. Dick, R. Joseph, and Li Shang. Power, Thermal, and Reliability Modeling in Nanometer-Scale Microprocessors. *Proceedings of the IEEE International Symposium on Microarchitecture (MICRO)*, 27(3):49–62, may-june 2007.

[19] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *Proceedings of ACM SIGGRAPH*, pages 777–786, 2004.

[20] Prashanth P. Bungale, Swaroop Sridhar, and Vinay Krishnamurthy. An Approach to Heterogeneous Process State Capture/Recovery to Achieve Minimum Performance Overhead During Normal Execution. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–26, 2003.

[21] Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs. In *Proceedings of the IEEE/IFIP International Workshop on Rapid System Prototyping (RSP)*, pages 34–40, 2007.

[22] Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. Communication-Aware Mapping of KPN Applications onto Heterogeneous MPSoCs. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 1262–1267, 2012.

[23] Weijia Che and Karam S. Chatha. Unrolling and Retiming of Stream Applications onto Embedded Multicore Processors. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 1272–1277, 2012.

[24] Guangyu Chen, Feihui Li, and Kandemir Mahmut. Compiler-directed Application Mapping for NoC-based Chip Multiprocessors. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 155–157, 2007.

[25] Jianjiang Cheng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 754–759, 2008.

[26] Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha. Executing Synchronous Dataflow Graphs on a SPM-Based Multicore Architecture. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 664–671, 2012.

[27] Chen-Ling Chou and Radu Marculescu. Incremental Run-time Application Mapping for Homogeneous NoCs with Multiple Voltage Levels. In *Proceedings of the IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 161–166, 2007.

[28] David Clarke, Alexey Lastovetsky, and Vladimir Rychkov. Dynamic Load Balancing of Parallel Computational Iterative Routines on Platforms with Memory Heterogeneity. In *Proceedings of the Conference on Parallel Processing (Euro-Par)*, pages 41–50, 2011.

[29] IETF Request For Comments. `http://datatracker.ietf.org/doc/rfc4880/`, visited August 2013.

[30] Daniel Cordes, Michael Engel, Peter Marwedel, and Olaf Neugebauer. Automatic Extraction of Multi-Objective Aware Pipeline Parallelism Using Genetic Algorithms. In *Proceedings of the ACM International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 73–82, 2012.

[31] Intel Corporation. `http://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-datasheet.pdf`, 2013.

[32] William J. Dally and Brian Towles. Route Packets, Not Wires: Onchip Interconnection Networks. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 684–689, 2001.

[33] Michael J. Donahoo and Kenneth L. Calvert. *TCP/IP Sockets in C: Practical Guide for Programmers*. Morgan Kaufmann, 2009.

[34] Paul Dubrulle, Stéphane Louise, Renaud Sirdey, and Vincent David. A Low-overhead Dedicated Execution Support for Stream Applications

on Shared-memory CMP. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, pages 143–152, 2012.

[35] Sahar Foroutan, Abbas Sheibanyrad, and Frederic Petrot. Cost-efficient Buffer Sizing in Shared-memory 3D-MPSoCs using Wide I/O Interfaces. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 366–375, 2012.

[36] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness.* W. H. Freeman and Co., 1979.

[37] M. Geilen, T. Basten, and S. Stuik. Minimising Buffer Requirements of Synchronous Dataflow Graphs with Model Checking. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 819–824, 2005.

[38] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, pages 3–14, 2001.

[39] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[40] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware computing in the angstrom processor. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 259–264, 2012.

[41] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`, 2001.

[42] Susan Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[43] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss,

T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 108–109, 2010.

[44] M. C. Huebscher and J. A. McCann. A Survey of Autonomic Computing Degrees, Models, and Applications. *ACM Computing Surveys (CSUR)*, 40(3):7:1–7:28, August 2008.

[45] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing âĂŤ Degrees, Models, and Applications. *ACM Computing Surveys (CSUR)*, 40(3):7:1–7:28, August 2008.

[46] Intel Threading Building Blocks Reference Manual. `http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm#reference/reference.htm`, 2013.

[47] ITRS. The International Technology Roadmap for Semiconductors. `http://www.itrs.net/`, 2012.

[48] Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, and Jörg Henkel. Runtime resource allocation for software pipelines. In *Proceedings of the ACM/EDAA 16th International Workshop on Software and Compilers for Embedded Systems*, pages 96–99, 2012.

[49] Janmartin Jahn, Sebastian Kobbe, Santiago Pagani, Jian-Jia Chen, and Jörg Henkel. Work in progress: Malleable software pipelines for efficient many-core system utilization. In *Proceedings of the 6th Intel Many-core Applications Research Community (MARC) Symposium*, pages 30–33, 2012.

[50] Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen, and Jörg Henkel. Runtime resource allocation for software pipelines. *ACM Transactions on Parallel Computing (submitted)*, 2013.

[51] Wooyoung Jang and David Z. Pan. A3MAP: Architecture-aware Analytic Mapping for Network-on-Chip. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 523–528, 2010.

[52] Janmartin Jahn and Jörg Henkel. Pipelets: Self-Organizing Software Pipelines for Many-Core Architectures. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages 1530–1521, 2013.

[53] Janmartin Jahn and Mohammad Abdullah Al Faruque and Jörg Henkel. CARAT: Context-Aware Runtime Adaptive Task Migration for Multi Core Architectures. In *Proceedings of the IEEE/ACM Design Automation and Test in Europe Conference (DATE)*, pages 515–520, 2011.

[54] Janmartin Jahn and Santiago Pagani and Jian-Jia Chen and Jörg Henkel. MOMA: Mapping of Memory-intensive Software-pipelined Applications for Systems with Multiple Memory Controllers. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 508–515, 2013.

[55] Janmartin Jahn and Santiago Pagani and Sebastian Kobbe and Jian-Jia Chen and Jörg Henkel. Optimizations for Configuring and Mapping Software Pipelines in Many Core Systems. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, 2013.

[56] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck Identification and Scheduling in Multithreaded Applications. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, 2012.

[57] Olivera Jovanovic, Peter Marwedel, Iuliana Bacivarov, and Lothar Thiele. Mamot: Memory-aware mapping optimization tool for mpsoc. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 743–750, 2012.

[58] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[59] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[60] Khronos OpenCL 1.2 Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, 2012.

[61] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *Proceedings of the IEEE International Conference on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[62] Kevin Klues, Barret Rhoden, Andreew Waterman, and Eric Brewer. Processes and Resource Management in a Scalable Many-Core OS. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[63] K. Knobe and C. Offner. TStreams: A Model of Parallel Computation (Preliminary Report). Technical report, HP Cambridge Research Lab, 2005.

[64] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: Distributed Resource Management for On-Chip Many-Core Systems. In *Proceedings of the IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, 2011.

[65] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 81–102, 2003.

[66] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 239–248, 2009.

[67] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1:323–337, 1992.

[68] ”Glenn Leary, Weijia Che, and Karam S. Chatha”. ”System-level Synthesis of Memory Architecture for Stream Processing Sub-systems of a MPSoC”. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 672–677, 2012.

[69] Edward A. Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[70] Haeseung Lee, Weijia Che, and Karam Chatha. Dynamic Scheduling of Stream Programs on Embedded Multi-Core Processors. In *Proceedings of the ACM International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 93–102, 2012.

[71] Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. An OpenCL Framework for Homogeneous Manycores with No Hardware Cache Coherence. In *Proceedings of the IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 56–67, 2011.

[72] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 53:1–53:11, 2007.

[73] Tuo Li, Ambrose, Jude Angelo, and Sri Parameswaran. Fine-Grained Hardware/Software Methodology for Process Migration in MPSoCs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 508–515, 2012.

[74] Mario Lodde, Jose Flich, and Manuel E. Acacio. Heterogeneous NoC Design for Efficient Broadcast-based Coherence Protocol Support. In *Proceedings of the 2012 IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 59–66, 2012.

[75] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, 2009.

[76] Radu Marculescu, Umit Y. Ogras, Li-Shiuan Peh, and Natalie Enright. Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives. *IEEE Proceedings of Transactions on Computer-Aided Design (TCAD)*, 28(1):3–21, 2009.

[77] Peter Marwedel, Jürgen Teich, Georgia Kouveli, Iuliana Bacivarov, Lothar Thiele, Soonhoi Ha, Chanahee Lee, Qiang Xu, and Lin Huang. Mapping of Applications to MPSoCs. In *Proceedings of the ACM International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 109–118, 2011.

[78] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram R. Vangal, Nitin Borkar, Gregory Ruhl, and Saurabh Dighe. The 48-core scc processor: the programmer's view. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010.

[79] Nicolas Melot et al. Investigation of Main Memory Bandwidth on Intel Single-Chip Cloud Computer. `http://www.ida.liu.se/~chrke/papers/MARC3-Symposium-Ettlingen-2011-Avdic-updated.pdf`, 2011.

[80] Jean-Yves Mignolet, Vincent Nollet, Paul Coene, Diederik Verkest, D. Verkest, Rudy Lauwereins, Serge Vernalde, and R. Lauwereins. Infrastructure for Design and Management of Relocatable Tasks. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages 986–991, 2003.

[81] Asit K. Mishra, Onur Mutlu, and Chita R. Das. A Heterogeneous Multiple Network-on-Chip Design: An Application-aware Approach. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, 2013.

[82] Takashi Miyamori, Hui Xu, Takeshi Kodaka, Hiroyuki Usui, Turo Sano, and Jun Tanabe. Development of Low Power Many-Core SoC for Multimedia Applications. In *Proceedings of the IEEE/ACM Design, Automation, and Test in Europe Conference (DATE)*, pages 773–777, 2013.

[83] M Monteyne. RapidMind Multi-Core Development Platform. Technical report, RapidMind Inc., 2007.

[84] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8):4–8, April 1965.

[85] Christian Müller-Schloer. Organic Computing: on the Feasibility of Controlled Emergence. In *Proceedings of the IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 2–5, 2004.

[86] V. Narayanan and Y. Xie. Reliability Concerns in Embedded System Designs. *IEEE Transactions on Computers*, 39(1):118–120, 1 2006.

[87] Vijaykrishnan Narayanan, Ahmed Al Maashri, Michael Debole, Matthew Cotter, Nadhini Chandramoorthy, Yang Xiao, and Chaitali Chakrabarti. Accelerating neuromorphic vision algorithms for recognition. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 579–584, 2012.

[88] R. Nelson and A. Tantawi. Approximate Analysis of Fork/Join Synchronization in Parallel Queues. *IEEE Transactions on Computers*, 37(6):739–743, Jun 1988.

[89] Luis Nogueira and Luis Miguel Pinho. Server-based Scheduling of Parallel Real-Time Tasks. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, pages 73–82, 2012.

[90] Vincent Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet. Centralized Run-time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, pages 234–239, 2005.

[91] Peach open movie project. `http://bigbuckbunny.org`, visited August 2013.

[92] OpenMP 4.0 Specificactions RC2. `http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf`, visited May 2013.

[93] OpenMP Website. `http://openmp.org/wp/about-openmp/`, visited May 2013.

[94] OpenMPI Website. `http://www.open-mpi.org/software/ompi/v1.6/`, visited May 2013.

[95] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[96] Matthew Papakipos. The PeakStream Platform: high-productivity software development for multi-core processors. In *Proceedings of Windows Hardware Engineering Conference (WinHEC), Industry Papers*, 2007.

[97] Jacques A. Pienaar, Srimat Chakradhar, and Anand Raghunathan. Automatic Generation of Software Pipelines for Heterogeneous Parallel Systems. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2012.

[98] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Optimal Task Assignment in Multithreaded Processors: A

Statistical Approach. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 235–248, 2012.

[99] Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. Thread Scheduling for Multi-Core Platforms. In *USENIX HotOS*, 2007.

[100] Michael Richmond and Michael Hitchens. A new process migration algorithm. *SIGOPS Operating Systems Review*, 31(1):31–42, 1997.

[101] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23(3):555–565, July 1976.

[102] Pardip Kumar Sahu, Putta Venkatesh, Sunilraju Gollapalli, and Santanu Chattopadhyay. Application Mapping onto Mesh Structured Network-on-Chip using Particle Swarm Optimization. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 335–336, 2011.

[103] Hartmut Schmeck. Organic Computing - A New Vision for Distributed Embedded Systems. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 201–203, 2005.

[104] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin haeng Kang, and Lothar Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the IEEE/ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 71–80, 2012.

[105] Thomas Serre and Maximilian Riesenhuber. Realistic Modeling of Simple and Complex Cell Tuning in the HMAX Model, and Implications for Invariant Object Recognition in Cortex. Technical report, Massachusetts Institute of Technology, July 2004.

[106] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, 2013.

[107] Jonathan M. Smith. A Survey of Process Migration Mechanisms. *SIGOPS Operating Systems Review*, 22(3):28–40, 1988.

[108] Allan Snavely and Dean Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, 2000.

[109] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the ACM Special Interest Group for the Computer Systems Performance Evaluation Community (SIGMETRICS)*, pages 66–76, 2002.

[110] Jan Stender, Silvan Kaiser, and Sahin Albayrak. Mobility-based runtime load balancing in multi-agent systems. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 688–693, 2006.

[111] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Transactions on Computing in Science and Engineering*, 12(3):66–73, 2010.

[112] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 356–369, 2007.

[113] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight Communications on Intel's Single-Chip Cloud Computer Processor. *ACM SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

[114] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 98–589, 2007.

[115] William Thies and others. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 179–196, 2001.

[116] Dong Hyuk Woo and H.-H.S. Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *IEEE Computer*, 41(12):24–31, 2008.

[117] Yedlapalli, Praveen and Kultursay, Emre and Kandemir, Mahmut T. Cooperative Parallelization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 134–141, 2011.

[118] Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov. Modeling Adaptive Streaming Applications with Parameterized Polyhedral Process Networks. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, pages 116–121, 2011.

[119] Andreas Zwinkau and Victor Pankratius. Autotunium: An evolutionary tuner for general-purpose multicore applications. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 392–399, 2012.