Scalable and Distributed Resource Management for Many-Core Systems

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

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

genehmigte
Dissertation

von

Sebastian Kobbe

aus Karlsruhe

Tag der mündlichen Prüfung: 11.5.2015

Erster Gutachter: Prof. Dr. Jörg Henkel
Zweiter Gutachter: Prof. Dr. Wolfgang Schröder-Preikschat
Dritter Gutachter: Prof. Dr. Wolfgang Karl

Sebastian Kobbe
Zähringerstr. 18
76131 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 habe 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.

Sebastian Kobbe

Abstract (German Version)

Many-Core Systeme stellen die Forschung vor vielfältige neue Herausforderungen, darunter die Handhabung von sehr dynamischer und kaum voraussagbarer Rechenlast. Optimale Ressourcenverwaltung von Many-Core Systemen ist ein NP-vollständiges Problem. Die große Anzahl der zu verwaltenden Anwendungen und Kernen führt bei zentral agierenden Heuristiken, welche stets einen globalen Blick auf das gesamte System haben, zu Skalierbarkeitsproblemen. Die Ressourcenverwaltung selbst kann so zum Engpass werden, welcher wegen hohen Latenzen die erzielbare Leistung des Systems einschränkt.

Der Schwerpunkt dieser Arbeit liegt im Erreichen von Skalierbarkeit der Ressourcenverwaltung. Die Komplexität der Ressourcenverwaltung wird durch verteilt stattfindende lokale Optimierungen gehandhabt. Jeder Anwendung wird ein eigener lokaler Ressourcenmanager zugewiesen, welcher den Bedarf an Ressourcen der Anwendung gegenüber den anderen Anwendungen vertritt. Die lokalen Ressourcenmanager kommunizieren über ein, sich am „Contract-Net Protokoll“ und an sogenannten „Gossip Protokollen“ orientierendes, Protokoll miteinander und verhandeln lokal darüber, wie die Verteilung der Ressourcen der betroffenen Anwendungen anhand eines vorgegebenen Optimierungsziels verbessert werden kann.

Parallele Anwendungen, welche sich während der Laufzeit an die ihnen zugewiesenen Ressourcen anpassen können, erlauben die Aufteilung (und Umverteilung) der Kerne des Many-Core Systems unter den gerade gleichzeitig ausgeführten Anwendungen, so dass diese effizient ausgeführt werden können. Dabei ist es jedoch wichtig, vor einer Ressourcenzuweisung die Auswirkungen auf die erzielbare Leistung der jeweiligen Anwendung abschätzen zu können.

Im Rahmen dieser Arbeit wurde ein adaptives Anwendungs-Performance-Modell entwickelt, welches anhand einer parametrischen Beschreibung der Anwendung abschätzt, welche Performance die jeweilige Anwendung unter einer bestimmten Ressourcenzuweisung erreichen wird. Es reagiert dabei auf spontane Variationen in der Belastung des Systems und berücksichtigt die topologischen Eigenschaften der Ressourcen. Im Vergleich zu vorherigen Anwendungs-Performance-Modellen wird der durchschnittliche Schätzfehler von 14,7% auf 4,5% reduziert.

Die vorgestellten Ressourcenverwaltung kann verschiedene Optimierungsziele verfolgen, welche durch eine Zielfunktion vorgegeben werden. Unter diesen Optimierungszielen ist beispielsweise die Maximierung des Durchschnitts der Performance der jeweils gleichzeitig ausgeführten Anwendungen. Die Annäherung an das Ziel

erfolgt dabei in mehreren inkrementellen Optimierungsschritten. Um dieses Ziel zu erreichen wurden zwei Strategien entworfen. Die erste Strategie ist eine komplexe Strategie und betrachtet in jedem Optimierungsschritt ganze Bereiche auf dem Chip, bestehend aus mehreren Kernen. Dabei erreicht sie nach wenigen Schritten eine gute Verteilung der Ressourcen. Ein Nachteil der komplexen Strategie ist, dass auch wenn keine Verbesserungen erzielt werden können, stets der gleiche Berechnungs- und Kommunikationsaufwand nötig ist. Die zweite Strategie ist eine leichtgewichtige Strategie und berücksichtigt im Gegensatz dazu in jedem Optimierungsschritt nur einen einzelnen Kern. Es sind jedoch viele kleine Optimierungsschritte notwendig, um das gleiche Ergebnis der komplexen Strategie zu erreichen. Die Summe des Gesamtaufwandes ist dabei jedoch höher.

Um die Effizienz der Ressourcenverwaltung zu verbessern, werden daher beide Strategien kombiniert. Dabei wird Wissen über den Bedarf an Ressourcen der jeweiligen Anwendungen für eine adaptive Auswahl zwischen den beiden Strategien genutzt. Nur wenn es aussichtsreich ist, die komplexe Strategie zu verwenden und die Differenz aus benötigten und zugeteilten Ressourcen die komplexe Strategie rechtfertigt, wird diese eingesetzt. Andernfalls kommt die leichtgewichtige Strategie zum Einsatz, um die Verteilung der Ressourcen unter den Anwendungen graduell zu verbessern.

Andere Forschungsgruppen wurden durch den gewählten Ansatz inspiriert, eigene, verteilte Ressourcenverwaltungsverfahren zu entwickeln. Verglichen zu diesen aktuellen Forschungsergebnissen wird durch die adaptive Wahl der Strategie der Berechnungsaufwand der Ressourcenverwaltung im Durchschnitt um 67% und der Kommunikationsaufwand um durchschnittlich 69% reduziert.

Insgesamt erzielt die in dieser Arbeit entwickelte verteilte Ressourcenverwaltung eine vergleichbar gute Verteilung der Ressourcen unter den gleichzeitig ausgeführten Anwendungen, wie diese von zentral agierenden Heuristiken erreicht wird. Im Gegensatz zu diesen besteht jedoch keine Gefahr, dass die Ressourcenverwaltung selbst zum Engpass wird. Sie erreicht also die Skalierbarkeit, welche notwendig ist, um zukünftige Many-Core-Systeme zu verwalten.

Abstract (English Version)

Many-core systems provide researchers with important new challenges, including the handling of very dynamic and hardly predictable computational loads. Managing the resources of many-core systems optimally is a NP-complete problem. The large number of applications and cores causes scalability issues for centrally acting heuristics, which always must retain a global view of the entire system. Resource management itself can become a bottleneck which – due to high computation and communication latencies – limits the achievable performance of the system.

The focus of this work is to achieve scalability of resource management. The complexity of resource management is handled through distributed local optimizations. A local resource manager is assigned to each application and represents the applications' resource demands in relation to other applications. The local resource managers communicate using a protocol which is based on the "contract-net" protocol and "gossip protocols". The resource managers negotiate locally on how the distribution of the resources of the affected applications can be improved using a predetermined optimization objective.

Parallel applications that can adapt to the resources allocated to them during runtime allow allocating (and reallocating) the cores of the system among the concurrently executing applications, leading to efficient system utilization of the available resources. However, it is essential to estimate the impact of resource (re-)allocation on the achievable performance of the application before allocating resources.

In this work, an adaptive application performance model was developed that estimates, based on a parametric description of the application, which performance will be reached by the application under a given resource allocation. The application performance model reacts to spontaneous variations in the load on the system and takes the topological properties of the resources into account. Compared to previous application performance models, the average estimation error is reduced from 14.7% down to 4.5%.

The presented resource management is able to optimize for different optimization goals, given through an objective function. Among these optimization goals is the maximization of the average performance of the concurrently running applications. Two strategies have been designed to realize an incremental, step-wise optimization of the objective function. The first strategy is a complex strategy and considers larger areas on the chip, consisting of several cores. This way, after a few steps, a

good allocation of resources is achievable. A disadvantage of this strategy is that even if no improvements can be achieved, the same computation and communication overhead is required. In contrast, the second strategy is a low-effort strategy and only takes a single core into account in each optimization step. However, there are many small optimization steps necessary to achieve the same result as the complex strategy, resulting in an overall higher effort if only the second strategy is used.

Therefore, in order to improve the efficiency of resource management, both strategies are combined. Knowledge about the resource demands of the respective applications is used for an adaptive selection between the two strategies. The complex strategy is only used if the difference between the required and allocated resources justifies its effort. Otherwise, the low-effort strategy is used to improve the distribution of resources among the applications gradually.

Other research groups have been inspired by the chosen approach to develop their own, distributed resource management. Compared to this recent research, the computational effort of the resource management presented in this thesis is reduced by an average of 67% and the communication overhead is reduced by an average of 69%.

Overall, the distributed resource management developed in this thesis achieves a comparably good distribution of resources among concurrently executing applications, similar to those achieved by centrally acting heuristics with global knowledge. In contrast to these, however, there is no risk that the resource management becomes a bottleneck itself. Thus, it reaches the scalability, which is necessary to manage future many-core systems.

List of Publications Contributing to this Thesis

[KBL+11] S. Kobbe, L. Bauer, D. Lohman, W. Schröder-Preikschat, and J. Henkel. DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, 2011.

[KBH15] Kobbe S., L. Bauer, and J. Henkel. Adaptive on-the-fly application performance modeling for many cores. In *IEEE/ACM 18th Design Automation and Test in Europe Conference (DATE´15)*, 2015.

[JPK+13] J. Jahn, S. Pagani, S. Kobbe, J.-J. Chen, and J. Henkel. Optimizations for Configuring and Mapping Software Pipelines in Many Core Systems. In *Design Automation Conference (DAC)*, 2013.

List of Further Publications

[HZZ+14] J. Heisswolf, A. Zaib, A. Zwinkau, S. Kobbe, A. Weichslgartner, J. Teich, J. Henkel, G. Snelting, A. Herkersdorf, and J. Becker. CAP: Communication aware programming. In *Design Automation Conference (DAC)*, 2014.

[JKP+12] J. Jahn, S. Kobbe, S. Pagani, J.-J. Chen and J. Henkel. Work in Progress: Malleable Software Pipelines for Efficient Many-core System Utilization. In *MARC Symposium at ONERA'2012*, 2012.

[HHB+12] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R.K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, S. Kobbe. Invasive Manycore Architectures. In *17th Asia and South Pacific Design Automation Conference (ASP-DAC'12)*, 2012

List of Supervised Student Projects

[TKH11] Jean Paul Tsague. Untersuchung von optimalen Schedules für variabel-parallele Anwendungen. Studienarbeit, Karlsruhe Institute of Technology, 2011.

[SKH11] Yuanyuan Song. Hardware Emulation Platform for future On-Chip Many-Core Systems. Diplomarbeit, Karlsruhe Institute of Technology, 2011.

[PKH12] Dennis Pahl. Distributed Resource Management on the Intel Single-Chip Cloud Computer. Diplomarbeit, Karlsruhe Institute of Technology, 2012.

[SKH12] Florian Tobias Schandinat. Distributed Resource Management on a On-Chip Many-Core System Demonstration Platform. Bachelor Thesis, Karlsruhe Institute of Technology, 2012.

[KKH13] Roman Klöpfer. Taskgraph basiertes Anwendungsmodell mit heuristischer Erkennung von charakteristischen Anwendungsphasen. Studienarbeit, Karlsruhe Institute of Technology, 2013.

[DKH13] Julian Dobrinkat. Simulative Design Space Exploration of Resource Management Approaches for Many-Core Systems. Diplomarbeit, Karlsruhe Institute of Technology, 2013.

[BKH14] Mario Bertolutti. Cycle-Accurate Evaluation of Multi-Agent Resource Management in Distributed Many-Core Systems. Bachelor Thesis, Karlsruhe Institute of Technology, 2014.

[KKH14] Kouassi Sepenou Klousseh. Analytical Resource-Aware Multi-Application Mapping to Many-Cores. Diplomarbeit, Karlsruhe Institute of Technology, 2014.

Contents

1. Introduction

Based on early advances in integrated circuit technology, G. Moore predicted an exponential growth of the number of transistors that could be integrated into a single integrated circuit [M+65] – often referred to as Moore's Law. Until today, this growth continued as predicted and allowed to design complex processor architectures. Together with advances in technology scaling that allowed increasing the clock frequency of these processors, computer systems became very powerful while only using a single processor core. However, physical changes in technology around the 90nm scale lead to constraints such as the power consumption of these cores combined with the high power densities [Sha07] that prohibit significant further increases in the clock frequency. The complexity of the architecture of individual processor cores has reached a level at which further improvements generate diminishing returns, i.e. the traditional ways to increase the performance of individual cores seem to have reached their limit.

One possibility to increase the overall system performance without violating physical limitations is to integrate multiple (potentially simple) processor cores in a single chip and to exploit parallelism in the executed application software. As the highest performance per watt and per chip area is achieved using less complex processor cores [Sha07], many-core systems are emerging. Accordingly, the International Technology Roadmap for Semiconductors (ITRS) [ITR13] predicts systems with thousands of cores for the next decade. Today, for example the Kalray MPPA [dDAB+13] processor already integrates 256 compute cores per chip, and up to 1024 cores per chip announced for 2015. Tilera's current Gx-Processor family [Til14] consists of up to 72 cores per chip. Instead of using a huge centralized interconnect, the individual cores communicate by using a Network-on-Chip (NoC) [Bor07].

An extensive number of cores in a many-core system poses the challenge of efficient system utilization. Mapping multiple applications to these cores and adapting each application to the allocated cores (e.g. by choosing a different algorithmic implementation or by increasing/decreasing the degree of parallelism) is key for an efficient utilization of the computation resources in many-core systems. Without proper resource management strategies, additional cores added to a many-core system will not automatically lead to a higher performance.

Figure 1.1 shows different resource allocations for four applications A_i, A_j, A_k, and A_l and the resulting overall system performance. In the example, A_l is capable of efficiently utilizing large numbers of cores. Allocating the same number of cores to

all applications (i.e. a *fair share* allocation, Figure 1.1a) typically results in a worse overall performance than allocating just the *right* resources to the *right* application (Figure 1.1b)). Over-saturating A_l at the cost of the other applications A_i, A_j, and A_k sacrifices system performance (Figure 1.1c). Therefore, to select the *right* resources, it is necessary to estimate or know the application performance under different resource allocations when deciding the application mapping.

In static scenarios with a fixed number of applications that do not change their resource demands at runtime, offline mapping of applications to cores can often provide optimal solutions. However, in situations with dynamic workloads in which e.g. the execution of a new application can be started at any time [CCD+08], mappings cannot be predetermined and need to be decided online.

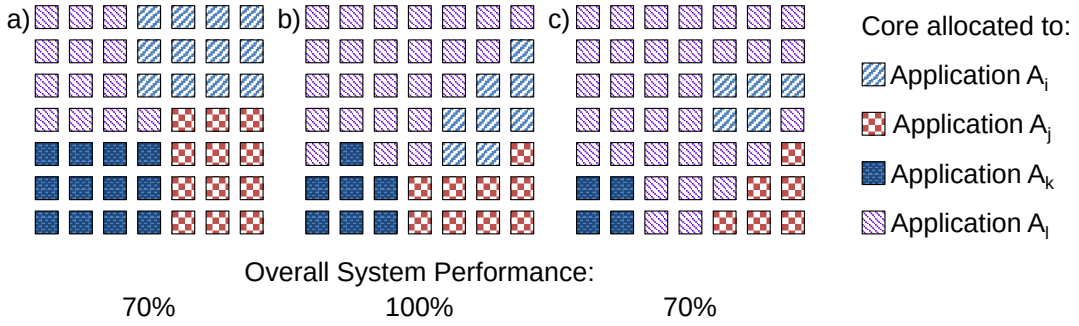

Figure 1.1.: Sketch on how the overall performance of a many-core system is influenced by the application mapping

A promising approach to efficiently utilize the available cores in dynamic scenarios is the principle of *malleable applications* ([TWY92], see Section 2.2) that are able to adapt the degree of parallelism to the number of allocated cores dynamically. This means that such a malleable application A_i is able to fulfill the same functionality on a variable number of cores and is designed in a way that allows it to *enlarge* or *shrink* the set of cores C_{A_i} that application A_i uses at runtime [KKD02]. However, each application's ability to increase its speedup (i.e. the relative performance compared to an execution on only one core) with the allocated cores varies. Therefore, select the *right* cores for each application, models that allow estimating the application performance on a certain set of cores before actually allocating these cores to the application are required.

Systems with hundreds or thousands of cores integrated on a single chip [ITR13] span a huge solution space that grows factorial with the number of cores [MMCM07]. The problem of optimal mapping of parallel applications to cores is known to be NP-complete [CGJ78]. The latency of deciding the application mapping, even when using fast heuristics, grows with the system size. Decision latencies in the order of seconds or even minutes limit the adaptation of the application mapping to dynamic workload situations significantly and prohibit an interactive utilization.

Hence, the problem of scalability of runtime resource management is of significant relevance. If no new paradigms are developed, complex future many-core systems will suffer from low efficiency since these systems will tend to spend large portions of their communication and computation capacities with managing their own resources instead of employing the resources for efficient application execution.

This results in two key challenges for efficient many-core system utilization: a) application performance estimation that allows selecting the *right* resources for each application and b) the scalability of runtime resource management. Both are presented in detail in the following Sections.

1.1. Application Performance Estimation

The performance of an application not only depends on the number of cores allocated to the application but also on their topological location on the chip. For example, the work presented in [MBS+05] has shown a 42% reduction of application execution time by improving the application mapping. To be able to select the right cores dynamically, accurate performance estimates are necessary (see Figure 1.2). These estimates rely on so-called application performance models.

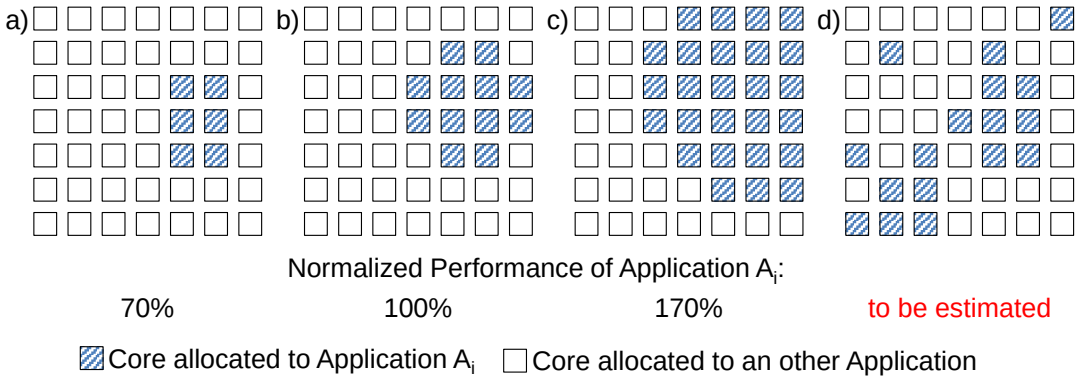

Figure 1.2.: Estimating the performance of an application A_i given a set of cores C_{A_i}

In some cases, it is possible to base these estimates solely on static application knowledge determined by offline analysis. However, in general this is not accurate, especially if the dynamic behavior of the application heavily depends on input data, e.g. a robotic vision application [PSK+12] that tracks and identifies objects. The more objects are recognized in the scene, the higher the potential parallelism is in the application which allows to efficiently utilize larger numbers of cores. Additionally, the performance of an application depends on the system architecture it is executed on (e.g. the ratio of computation to communication performance [HZQ+13, LNC13]) and the system load caused by concurrently executing applications.

As the number and kind of concurrently executing applications is not known a priori, in most situations estimations solely based on offline analysis will result in inaccurate estimates and thus in disproportionate application mappings. A high number of estimates (see Section 5.3) is calculated at runtime to find a suitable allocation of cores to each application whenever a change in the workload triggers a new application mapping. This requires the performance estimation to be lightweight – otherwise, the latency of runtime resource management negates the improved application mapping quality.

1.1.1. Motivational Example

Consider an example application A_x that only executes a sequential section of code followed by a parallel section, followed by a sequential section. This represents the traditional fork-join programming style, e.g. reading input data, parallel processing, writing output, see Figure 1.3. Therefore, execution time depends on the time $T_{forward}$ required to forward the input data from the core that processed the input to the cores that process the data and $T_{collect}$ to collect the results from the respective cores. The total execution time is $T_{input} + T_{forward} + T_{process} + T_{collect} + T_{output}$. The communication delays depend on the selection of cores in C_{A_x} (see Section 3.1). Depending on the communication latencies between the cores in C_{A_x}, the available bandwidth of the NoC, and the amount of transferred data. The delays $T_{forward}$ and $T_{collect}$ significantly limit the speedup of the application [HZQ+13]. Bandwidth can be consumed either by concurrent communication of other applications or by communication of different tasks of the same application.

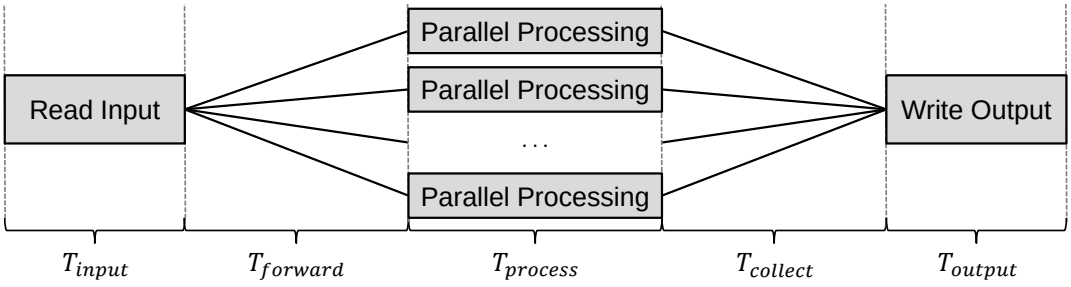

Figure 1.3.: Components of the total execution time of a hypothetical application A_x

To select the right cores C_{A_x}, accurate estimates of the achievable performance are required. In this thesis, the speedup of the application compared to the execution on a single core is used as relative performance metric. However, determining the speedup $S_{A_i}(C_{A_i})$ while considering the topological location of the cores in C_{A_i} is in general computationally more intensive than $S^{\#}_{A_i}(|C_{A_i}|)$ where only the number of cores is considered. The former would entail an actual mapping of the tasks while the latter would always be an estimate, based e.g. on the values obtained by offline profiling of the application. Due to the large number of different subsets of

cores, it is not feasible to pre-compute and store all combinations. Even when only considering different subsets of cores with an identical number of cores ($|C_{A_i}| = |C'_{A_i}|$ but $C_{A_i} \neq C'_{A_i}$), there are approximately 10^{47} possible ways to select 40 different cores out of 256 cores (binomial coefficients, Equation (1.1)). The metric $S^{\#}_{A_i}(|C_{A_i}|)$ estimates the same speedup for all these subsets, but actually they differ depending on the topological location of the cores. Therefore, ignoring the topology of the resources assigned to each application leads to inefficient application mappings.

$$\binom{N}{|C_{A_i}|} = \binom{256}{40} \approx 10^{47} \tag{1.1}$$

Accurately calculating $S_{A_i}(C_{A_i})$ by using the scheduling heuristic presented in [THW02] requires several milliseconds (validated by measurements of an implementation on an Intel i5-2500 running at 3.3 GHz), depending on the number of tasks in the application task-graph and the number of cores in C_{A_i}. However, many estimations have to be calculated for each mapping decision – the hill-climbing heuristic used in the evaluation (see Section 5.3.1) considers more than 10,000 combinations of C_{A_i} to find a good distribution of 256 cores to 10 different applications (and even 62,000 combinations to distribute 256 cores to 20 applications). Optimistically assuming that there is no additional overhead and that the average computation time of determining $S_{A_i}(C_{A_i})$ is just 1 ms, this results in a total execution time of 10-60 seconds to determine the resource allocation for 10-20 applications on a 256-core system. This is an unacceptable high latency, e.g. when starting a new application for interactive utilization.

1.2. Resource Management

Multiple applications $\{A_1, A_2, \ldots, A_M\}$ are executed concurrently. With the ability to estimate the performance of each application for potential resource allocations introduced in the previous Section, assigning resources to each application is an optimization problem defined by an optimization goal like the maximization of the average application performance or the minimization of the time required to complete all concurrently executing applications, as described in Section 5.2.

The goal of resource management is to (re-)allocate a subset of cores $C_{A_i} \subseteq \mathcal{C}$ to each application A_i such that the chosen optimization goal is reached and that no core is assigned to two applications at the same time, i.e. for all A_i, A_j with $i \neq j$, $C_{A_i} \cap C_{A_j} = \emptyset$. To achieve the optimization goal in dynamic workload scenarios, the application mapping is adapted whenever a new application is about to start execution, an application terminated, or the resource demands of one of the momentary executing applications change. This results in frequent opportunities for application (re-)mapping in dynamic workload scenarios. Considering the size of the solution space, optimal solutions can not be derived at runtime.

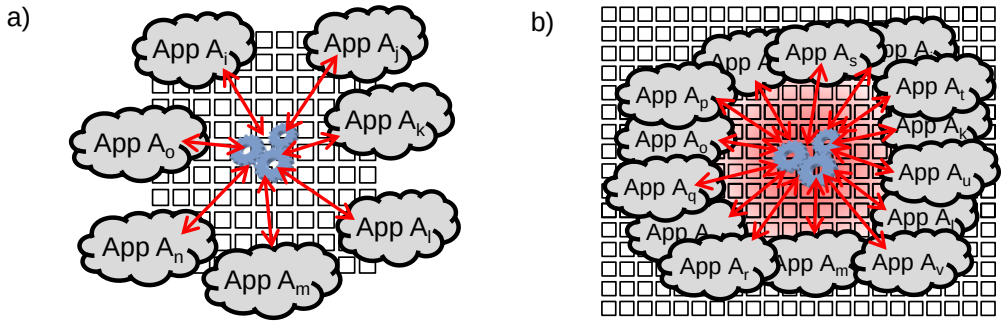

Figure 1.4.: Scalability issues of centralizes resource management under growing numbers of applications and cores

However, even fast heuristics like the ones presented in Section 5.3 take too long (i.e. multiple seconds) to react on frequent changes in the workload of multiple concurrent applications. Such high latencies, however, negate the benefits of frequent adaptations of the application mapping. Additionally, the communication infrastructure surrounding the core a centralized resource manager is located on is heavily utilized which may lead to additional latencies trough link congestion (see Figure 1.4) and frequent adaptations in turn cause increasing overhead and therefore exacerbate scalability problems. The computational complexity and the ongoing communication may often lead to computation/communication bottlenecks which break the scalability.

Scalability in general is the ability of a technique to stay within feasible computation constraints under growing problem sizes. Scalability for resource management is crucial with respect to the computation overhead (i.e. the amount of computation needed to manage the resources and the involved latencies) as well as the induced communication volume. The challenge in resource management is to achieve high application mapping quality with respect to the optimization goal while at the same time keeping the associated overhead and the latencies of computing the application mapping low.

1.2.1. Motivational Example

To motivate the benefits of frequent adaptations of the application mapping at runtime, the NWChem application package [KAB+00] for large-scale simulations of chemical and biological systems was chosen as a real-world example. However, similar examples for runtime varying resource demands of an application can be found in other domains [CCD+08], too.

Simulations performed with NWChem comprise multiple computation kernels. When executed on the same number of cores the computation kernels show varying speedups and thus a different efficiency, which is defined as speedup in relation to

the number of used cores, see Equation (3.4). The number of cores allocated to one computation kernel might not be sufficient to exploit the parallelism of the next kernel or it might lead to an inefficient utilization.

An example from NWChem is a parallel linear algebra calculation consisting of three computation kernels: calculation of Eigenvectors (EV), Householder reduction (HH), and back-transformation (BT). Figure 1.5 shows the speedup of each kernel, as presented in [KAB+00]. The HH kernel does not scale beyond 32 cores, whereas the EV and BT kernels (that are executed before and after HH, respectively) can efficiently utilize more cores.

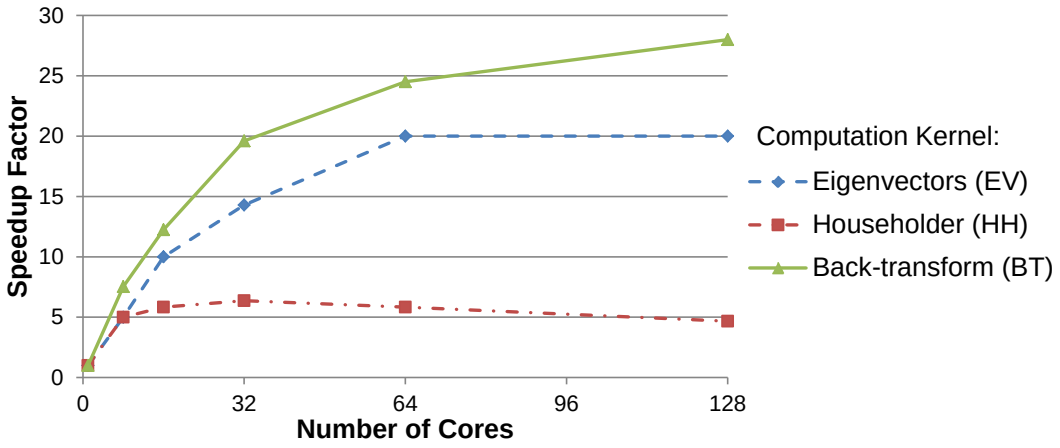

Figure 1.5.: Speedup of the computation kernels of a parallel linear algebra calculation. The kernels are repetitively executed in sequence ($EV_1 \rightarrow HH_1 \rightarrow BT_1 \rightarrow EV_2 \rightarrow \dots$) and vary in their resource demands (based on measurements presented in [KAB+00])

When multiple applications are executed on a many-core system, the overall system performance highly depends upon the application mapping. Figure 1.6 shows in a simplified example how the total execution time of two concurrent applications benefits from adapting the application mapping according to the computation demands of the applications. The improvement in application execution efficiency comes at the cost of multiple invocations of the resource management. Note that in the example only two applications are shown. For a larger number of applications, far more opportunities for re-mapping exist.

1.2.2. Achieving Scalability

Scalability may be achieved by moving from centralized resource management to distributing the resource management throughout the chip and by using only local information of the system state in order to perform local application mapping optimizations [KBL+11, ATBS13, Wei99].

Figure 1.6.: Execution time of two applications a) with and b) without considering the varying resource demands at runtime. The respective momentary application mapping quality is given by the combined application speedup. Frequent adaptations result in an overall higher application mapping quality and a reduced execution time of both applications

In this thesis one local resource manager is employed per application for managing the cores C_{A_i} of the respective application A_i, and communicating the related resource demands to the other resource managers with the goal to achieve a chip-wide coordinated resource management. Compared to centralized resource management, this improves scalability for three main reasons:

- The computational complexity of resource management is reduced, as only a subset of the system resources is considered for decision making and therefore the problem size gets decoupled from the system size.
- The resource management becomes inherently parallel.
- The communication infrastructure gets equally utilized. The individual resource managers do not demand system-wide synchronization or knowledge of the global system state. Communication takes place locally to avoid communication bottlenecks to ensure scalability.

Figure 1.7 visualizes the scalability advantage of distributed resource management. There is no centralized bottleneck, so the system may expand indefinitely without the addition of supporting resources (other than the resource managers themselves).

In literature such local computational entities are often referred to as *Agents*. Multiple interacting Agents form a *Multi-Agent-System* [Wei99] (see Section 2.3). In the reminder of this thesis the terms “local resource manager” and “Agent” will be used interchangeably.

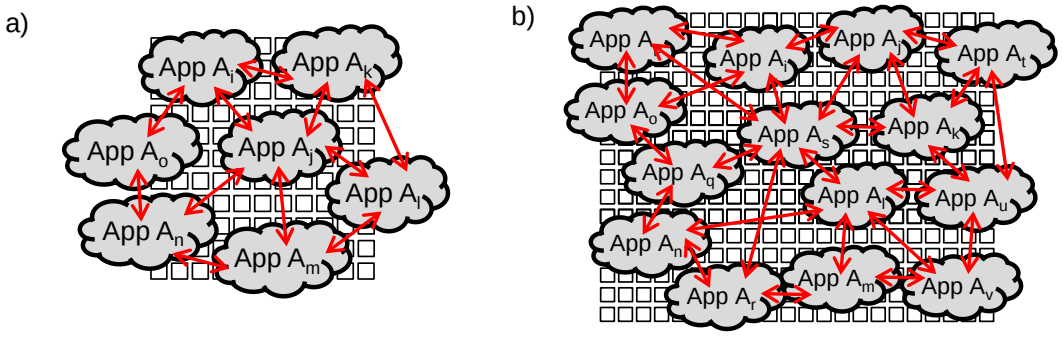

Figure 1.7.: Scalability of distributed resource management: the system may expand indefinitely without the addition of supporting resources (other than the resource managers themselves)

1.3. Thesis Contribution

The contribution of this thesis is twofold: first, an adaptive **on-the-fly application performance model** that considers the topological properties of the cores allocated to an application in NoC based many-core systems is presented. Secondly, the scalability issues of online resource management for on-chip many-core systems are addressed by means of a fully **distributed resource management**:

- The adaptive on-the-fly application performance model (see Chapter 4) considers the topological properties of the cores allocated to an application in NoC based many-core systems. It uses a simple metric of C_{A_i} that can easily be determined to estimate the achievable performance based on the lower bound and the upper-bound performance of the application. To handle highly dynamic behavior of workloads not known a priori the application performance model is continuously adapted at runtime. Compared to static application models which do not consider the topological properties, the estimation accuracy is improved while the computation effort stays within feasible bounds for online application mapping decisions.
- The distributed resource management presented in Chapter 5 is able to flexibly react to changes of the resource demands of applications by adaptively selecting the employed resource management strategy, whereof two are developed in the scope of this thesis. The resource management is able to find a good initial mapping of applications fast and to optimize the application mapping gradually with a very low overhead. It uses the concept of a Multi-Agent-System [Wei99] to achieve scalability by focusing on local decision making based locally available information.
- Both contributions work hand in hand (see Figure 1.8 and Chapter 6) to improve the quality of resource management and to reduce the necessary computational and communicational overhead. Compared to centralized resource management,

the application mapping latency is reduced in order to support the efficient execution of highly dynamic workloads on many-core systems.

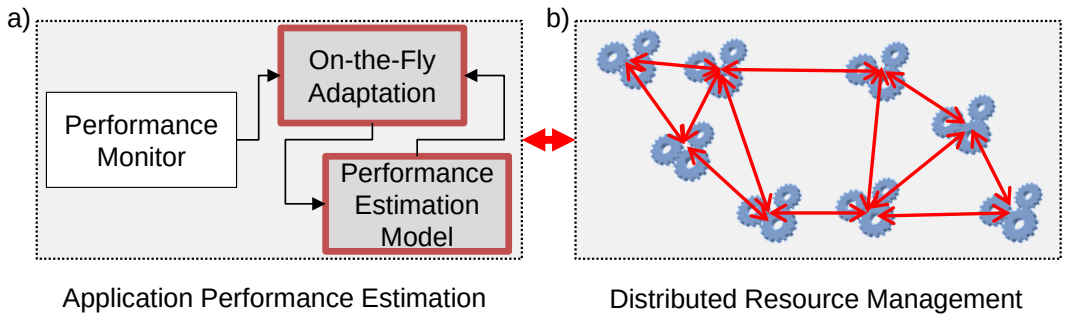

Figure 1.8.: Thesis contributions: a) adaptive on-the-fly application performance modeling and b) scalable distributed resource management for many-core systems

1.4. Thesis Outline

The reminder of this thesis is structured as follows: Chapter 2 first introduces historic, current and future many-core systems. As pointed out before, malleable applications are key for efficient many-core utilization. Therefore, Section 2.2 gives a broad overview on how malleability in an application can be realized. Section 2.3 presents the concepts of Multi-Agent Systems. Two selected protocol families ("The Contract-Net Protocol" and "Gossip Protocols") are described in detail as they inspired the protocols developed in this thesis. A brief overview on operating systems for many-cores concludes the Chapter.

Chapter 3 distills the information given in Chapter 2 into the models and assumptions used in this thesis. It presents the many-core platform model (Section 3.1), the application model (Section 3.2), and the functionality assumed to be provided by an operating system to facilitate resource management.

As motivated, resource management entails an estimation of the performance an application A_i achieves when executed on a certain set of cores C_{A_i}. Chapter 4 presents the adaptive on-the-fly application performance modeling developed in the scope of this thesis. The model (Section 4.4) is based on an empirical analysis (Section 4.3). Section 4.6 presents the on-the-fly adaptation of the model parameters and Section 4.7 evaluates the estimation accuracy and the introduced overhead.

In Chapter 5, the scalable distributed resource management for many-cores is presented. Section 5.1 begins with an overview over related work on resource management in different many-core domains. As the developed strategies are not limited to a specific optimization goal, different goals are presented in Section 5.2.

Examples for centralized resource management and their resulting latencies are shown in Section 5.3 to motivate the need for distributed resource management. Infrastructure required to realize the resource management presented in this thesis is described in Section 5.4.1. The two optimization strategies developed in the scope of this thesis, DistRM and a low-effort strategy are presented in Section 5.4.3 and Section 5.4.5, respectively. Finally, the adaptive strategy selection AStra that combines both is described in Section 5.4.6.

Chapter 6 presents an extensive evaluation of the different resource management strategies developed in the scope of this thesis (the low-effort strategy in Section 6.1, DistRM in Section 6.2, and AStra in Section 6.3), and State-of-the-Art distributed resource management in Section 6.4.2. They are compared to each other in Section 6.5, where Section 6.5.1 presents the average of several application scenarios and Section 6.5.3 shows the different behavior of the different resource management strategies for different application scenarios.

Finally, Chapter 7 concludes this thesis and gives an outlook on future work. Appendix A presents the simulation environment that was developed and used in this thesis, and Appendix B details the workload used for evaluation. Appendix C briefly shows the developed hardware platform that demonstrates the fully distributed resource management in a real implementation. Appendix D gives details on the implementation of the distributed resource management developed in the scope of this thesis on the Intel SCC.

2. Background

To lay the foundation of this thesis and to motivate the models and assumptions presented in Chapter 3, this chapter first introduces some historic and current many-core chips. Several ways how to realize *malleable applications* that can adopt their execution to the assigned resources, are presented in Section 2.2. As Multi-Agent Systems are a central building block of the resource management developed in the scope of this thesis, they are introduced and defined in Section 2.3. Two fundamental protocols traditionally used in Multi-Agent Systems that inspired the protocols designed in this thesis are presented. Section 2.4 gives a brief overview on current many-core operating systems and how they address the problem of resource management.

2.1. Many-Core Hardware Architectures

This Section introduces different many-core architectures. The platform model used in this thesis (presented in Section 3.1) is based on these architectures. Section 2.1.1 starts with a historic example the Transputer. The Intel many-core research platform "Single Chip Cloud Computer" is presented in Section 2.1.2. Two commercially available many-core platforms, the Tilera TILE-Gx processor family and the Kalray MPPA are presented in Section 2.1.3 and Section 2.1.4, respectively.

2.1.1. A historic example: The Transputer

While not exactly an *on-chip* many-core system, the Transputer (first announced in 1983 and released in 1984) [WS85] was one of the first systems to have multi-processing and parallelism as a key design feature. Technology of that time limited the number of transistors to be integrated in a single chip. Therefore a Transputer system consisted of many individual chips that contained one processing core each[1]. The different chips were connected by dedicated communication channels, able to form a 2d-mesh network, similar to to the networks on-chip seen in today's many-core systems. Each core has its own memory, communication takes place by message

[1] Appendix C presents a hardware demonstrator platform which runs a variant of the resource management developed in this thesis. Its hardware design, consisting of several micro-controllers, is heavily inspired by the Transputer.

passing. Theoretically, computing power of a Transputer system could simply be increased by adding additional Transputer chips. Figure 2.1 shows how multiple Transputer were connected to larger *many-core* boards. Multiple of these boards could be connected to create systems with hundreds or thousands of Transputer chips.

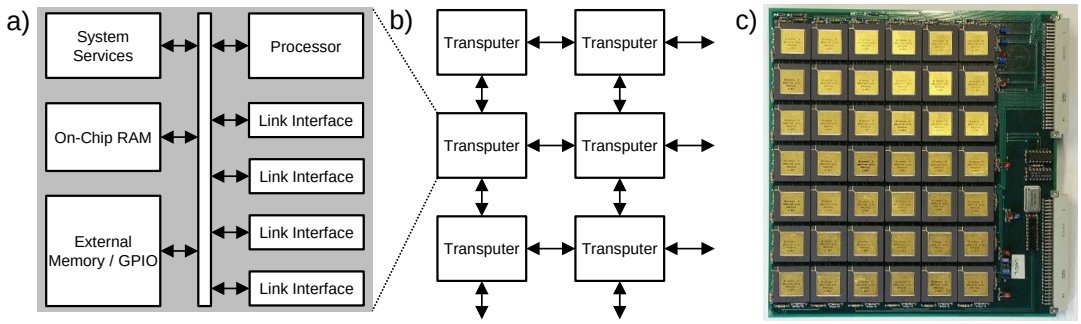

Figure 2.1.: Block diagram of a) each Transputer chip and b) the possible 2d-mesh interconnect of multiple Transputer chips. Subfigure c) shows a picture of the B0042 evaluation board containing 42 Transputer chips [May]

Hardware Profile "Transputer"

Number of cores:	$1 - \infty$
Core architecture:	16Bit (T2)
	32Bit (T4)
	64Bit (T8)
Core clock speed:	15 – 30 MHz
Cores per cache coherent domain:	1
Primary means of communication:	Message-Passing
Communication infrastructure topology:	arbitrary, e.g. 2d-mesh Network
Communication infrastructure bandwidth:	5 – 20 Mbit/s per link
	4 links per core
On-Chip memory per core:	4 kByte
Operating system support:	multiple, e.g. HeliOS or
	"Bare Metal" occam

The envisioned programming model of the Transputer was formalized in the programming language occam [May83] which allows to describe programs in the format of Communicating Sequential Processes (CSP) [Hoa78]. The Transputer contained a dedicated hardware scheduler. Processes were ready for execution once their input data arrived. Multiple Processes were transparently multiplexed on the individual chips without the need for a software scheduler in an Operating System to keep the overhead low.

Despite its visionary architecture, the Transputer did not succeed and failed to change the way of computing at its time. Still, it provoked new ideas in computer architecture, several of which appeared again or are starting to appear again.

2.1.2. Intel Single-Chip Cloud-Computer

The Intel Single-Chip Cloud-Computer (SCC) was introduced in 2009 and integrates 48 IA-32 x86 cores in a 6×4 2D-mesh network, i.e. two cores each are combined to a so-called tile that is connected to the Network-on-Chip. The chip was made available to several research institutes and academics. Thereby, Intel planed to gain a better understanding of how to schedule and coordinate the many cores of this experimental chip. Cores primarily communicate through message passing and there is no hardware-managed memory coherence for the globally addressable external memory [HDH+10]. Figure 2.2 shows the block diagram of the Intel SCC and a photograph of the die.

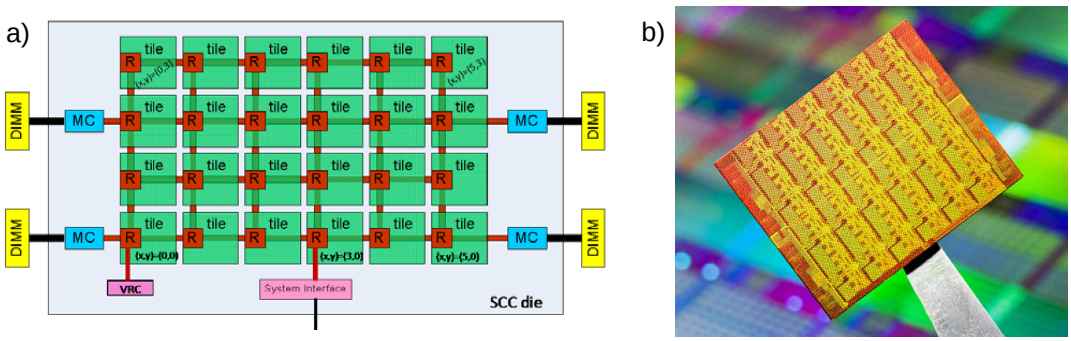

Figure 2.2.: Intel Single Chip Cloud Computer (SCC). Subfigure a) shows the block diagram and b) an actual picture of the fabricated die[HDH+10]

The Intel SCC can be operated in several modes: It is possible to execute a separate instance of the Linux Kernel [Fou] on each core. This allows to execute traditional software as it is common in cloud-computing or high-performance computing environments. Further, a "bare metal" runtime environment that allows to program the Intel SCC without the overhead of running Linux on each core is available. To take advantage of the dedicated message passing hardware integrated on the Intel SCC, a message passing library called RCCE was developed. RCCE is available in both modes of operation [MRL+10]. Additionally, the Barrelfish Operating System [BBD+09] – described in Section 2.4.2 – was ported to the Intel SCC as its design principles were a close match to the hardware design of the Intel SCC [PRB10].

The resource management [KBL+11] developed in the scope of this thesis has been implemented for the Intel SCC [PKH12]. A component based middleware that runs on top of Linux on each core of the SCC (see Appendix D) allows an efficient implementation and evaluation. Insights gained by evaluating the behavior

on the real hardware platform lead to improvements of the system level simulation environment (see Appendix A) as well as the resource management itself.

Hardware Profile "Intel SCC"

Number of cores:	48
Core architecture:	32Bit x86
Core clock speed:	1 GHz
Cores per cache coherent domain:	2
Primary means of communication:	Message-Passing
Communication infrastructure topology:	2d-Mesh Network
Communication infrastructure bandwidth:	64GB/s per link
	2TB/s bi-section bandwidth
On-Chip memory:	≈8MB
Operating system support:	Multiple Linux instances
	"Bare-Metal"
	Barrelfish

2.1.3. Tilera TILE-Gx Processor Family

The commercially available Tilera TILE-Gx processor family currently includes processors with 9, 16, 36, or 72 identical processor cores interconnected with an on-chip network. Systems with even more cores have been announced but are not available yet. Figure 2.3 shows the block diagram of the Tilera TILE-Gx72™ and a picture of the TILEncore-Gx72™ card which is suitable as an application accelerator or directly as a TILE-Gx development platform.

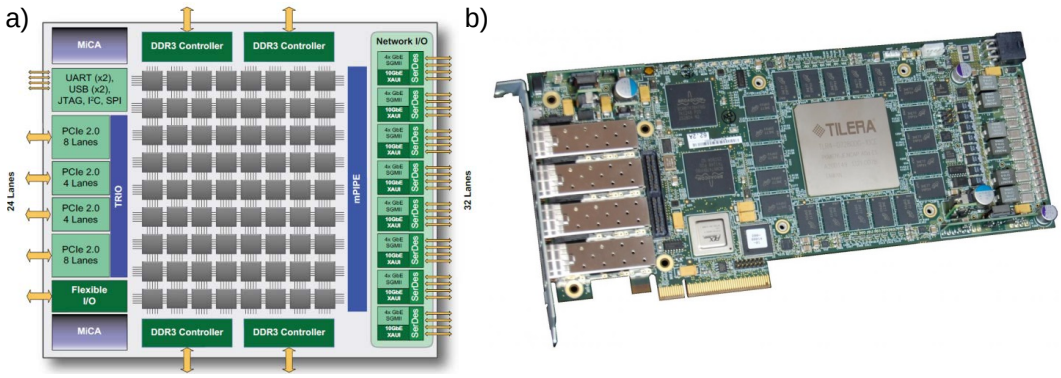

Figure 2.3.: a) Tilera TILE-Gx72™ Processor Block diagram and b) a picture of the TILEncore-Gx72™ card [Til14]

Each processor core has is own L1 and L2 cache, however there is hardware supported cache coherence which allows to execute standard symmetric multi-processing

(SMP) Linux on the TILE-Gx processors. Additionally, a "zero overhead Linux" variant is available, which allows the execution of almost unmodified applications without providing traditional operating system services like interrupt handling, etc. Similar to the Intel SCC, a "bare metal" operation allows to optimize for the highest possible performance. Any mode of operation can be spatially mixed on the TILE-Gx processors, i.e. a group of cores can run an instance of SMP Linux while other cores operate in bare metal mode and another group of cores runs a separate instance of SMP Linux [Til14].

The TileGX processor family is mostly used for high bandwidth network processing (e.g. load-balancing, filtering and monitoring), video processing (e.g. transcoding of video streams for different bandwidth in real-time), and cloud-computing.

Hardware Profile "Tilera TILE-Gx"

Number of cores:	9 – 72
Core architecture:	64 Bit "full-featured"
Core clock speed:	1.2 GHz
Cores per cache coherent domain:	all
Primary means of communication:	Shared-Memory Message-Passing
Communication infrastructure topology:	2d-Mesh Network
Communication infrastructure bandwidth:	>110 Tbps aggregate bandwidth
On-Chip memory:	23 MB
Operating system support:	Linux and "Bare Metal"

2.1.4. Kalray MPPA

The Kalray MPPA-256 is a many-core chip that integrates 256 user cores and 32 system cores. These cores are distributed across 16 compute clusters of 16+1 cores, and 4 quad-core I/O subsystems. Each compute cluster and I/O subsystem owns a private address space, while communication and synchronization between them is ensured by an on-chip network [dDAB+13, dDdML+13]. Each core is a 32-bit Very Long Instruction Word (VLIW) processor running at chip frequency. It may execute up to four "RISC like" instructions per cycle. With this architecture, the Kalray MPPA exploits instruction level parallelism, thread level parallelism (within compute cluster) and process level parallelism (between compute clusters).

The MPPA can be programmed in a data-flow description language, providing developers a solution for describing highly parallel applications without being constrained by the underlying hardware platform. This description is then used by a compiler to fully automate the mapping of tasks to the MPPA clusters and the data routing through the NoC. Similar to a general purpose graphics processing units

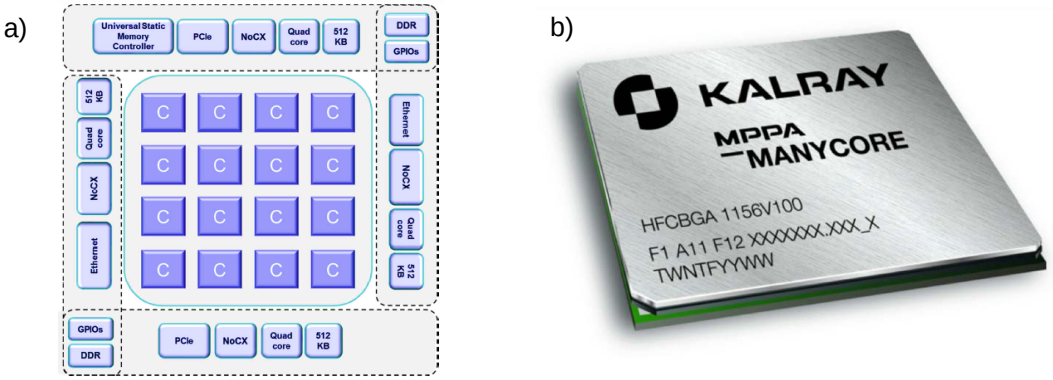

Figure 2.4.: Kalray MPPA 256 Core System. Subfigure a) shows the block diagram and b) a picture of the actual chip[dDAB+13]

(GPGPU) like the NVidia Tesla [LNOM08], the tool-chain is optimized to run only one application at a time and uses static mapping of application tasks to cores.

Additionally a POSIX like programming environment is available that offers multi-threading per compute cluster and inter-process communication through the Network-on-Chip to communicate and synchronize between the compute clusters.

Hardware Profile "Kalray MPPA-256"

Number of cores:	256
Core architecture:	32Bit VLIW
Core clock speed:	400 MHz
Cores per cache coherent domain:	16 compute cores 1 core dedicated for I/O
Primary means of communication:	Shared Memory Message-Passing
Communication infrastructure topology:	2D-wrapped-around torus
Communication infrastructure bandwidth:	up to 3.2 GB/s
On-Chip memory:	2 MB per compute cluster 32 MB total
Operating system support:	"POSIX"

2.2. Malleable Applications

After the previous Section 2.1 briefly introduced many-core architecturs, this Section focuses on the different kinds of parallel applications that may be executed on such many-core systems. In general, parallel applications are classified as follows [FR96]:

Rigid applications require a fixed number of cores. They can not start execution on less cores than required and they will not utilize additional cores (see Figure 2.5a). The resource manager has no means to adapt the resource allocation to the momentary workload of the system.

Moldable applications can execute over a wide range of cores. This number of cores has to be decided at the beginning of the execution of the application and stays constant until termination of the application execution (see Figure 2.5b). Applications following the SPMD[2] style are typically moldable. The resource manager can adapt the application to the momentary workload of the system only once and cannot react to changes in the workload.

Malleable applications are very flexible and can adapt to changes in the set of allocated cores at runtime (see Figure 2.5c). The resource manager has a high degree of freedom to adapt each application to the momentary workload of the system. For efficient adaptations, the resource manager should be aware of the current resource demands of the application. Malleable applications that inform the resource manager about their resource demands are sometimes also referred to as *evolving* applications and are the targeted kind of applications in this thesis.

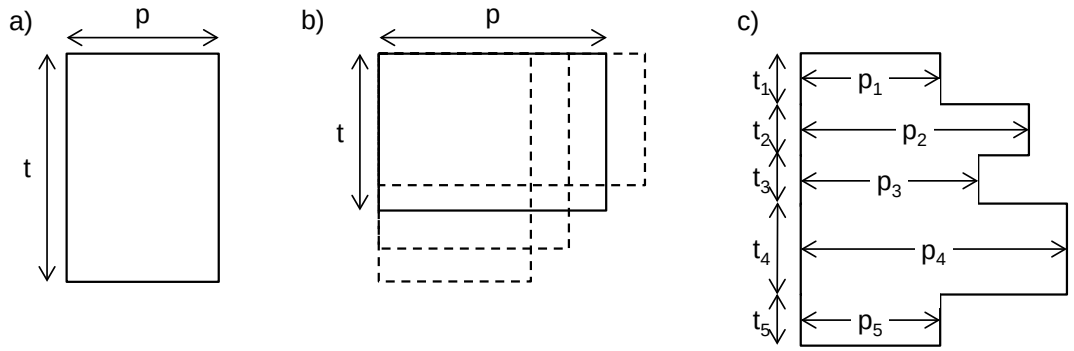

Figure 2.5.: Parallelism p over time t of a) rigid applications, b) moldable applications, and c) malleable applications. Adapted from [FR96]

Malleable pplications that can adapt themselves to changes in the set of allocated cores at runtime have proven to improve the system utilization and the average response time [KKD02]. This section now continues with a brief overview on how malleable applications may be realized.

The programmer of an application could either directly design the application to be malleable (e.g. using master/worker architectures with varying amounts of workers per master to distribute the work among the workers and/or use work-stealing techniques [BL94]) or use frameworks that support the automated creation

[2]Single Program, Multiple Data – The dominant style of parallel programming, where all processors use the same program, though each has its own data [Ata98], e.g. MPI applications

of malleable applications. Intels Threading Building Blocks [Rei07], for example, support the programmer to create malleable applications by automatically applying techniques similar to work-stealing to efficiently make use of varying numbers of available cores.

2.2.1. Master/Worker Paradigm

Most probably the most simple and intuitive approach for a malleable application is the master/worker paradigm. Sometimes, it is also referred to as master/slave paradigm or simply as task-queue [SV96]. To apply the master/worker paradigm, the problem must be divisible into multiple, smaller *jobs*. The master assigns theses jobs to its workers, collects their results and combines them to a solution for the original problem.

Allocating additional cores to the application allows the master to spawn additional workers such as the parallelism of the application is increased. Retrieving cores from the application is possible whenever a slave finished the execution of its assigned job. In the worst case, the application reconfiguration latency is determined by the execution time of the individual jobs. If the reconfiguration latency is the main concern, workers might simply be terminated before completing the execution of their job, which then is re-assigned to an other worker. However, the expected gain of the reduced latency should outweigh the additional overhead of re-computing the partially solved job. Migrating the state of the partial solution to an other worker [JAFH11] might also be an viable solution, at the cost of a significantly more complex design of the application.

The master/worker paradigm is broadly applicable and has be applied to several typical heuristics, e.g. parallel genetic algorithms [Ism04], and branch-and-bound [ANF03]. Another example for the master/worker paradigm are image-processing applications where the workers operate on independent regions of a larger image.

2.2.2. Adaptive MPI

Standard MPI programs divide the computation into N processes, one for each core. In contrast, adaptive MPI [HLK04] divides the computation into a large number of virtual processes, independent of the number of physical cores. Adaptive MPI includes a powerful runtime support system that takes advantage of the degree of freedom afforded by allowing it to assign virtual processes onto cores. It enables automatic load balancing and checkpointing. Most importantly, it has the ability to shrink and expand the set of processors used by a job at runtime. With other words: by using the Adaptive MPI runtime instead of a common MPI implementation, any application that relies on MPI can be made malleable. The virtual processes are programmed in MPI as before. Physical cores are no longer visible to the programmer,

as the responsibility for assigning virtual processes to physical cores is taken over by the runtime system.

Adaptive MPI builds on top of the CHARM++[KK93] processor virtualization system and uses its communication facilities and load balancing strategies. The work presented in [HZKK06] demonstrated, that the overhead of adaptive MPI is small and tolerable for most applications. Thus, the benefits of adaptivity and malleability come at only a small cost.

Adaptive MPI only balances the load of *one* application to a given allocation of cores. It does not decide which application should use which and how many cores. Adaptive MPI does not provide direct feedback to the resource manager to decide the resource allocation, however e.g. the (average) number of tasks ready for execution in the work queues might be an indicator to assign additional resources to an application. Otherwise, an application performance model like the one presented in Chapter 4 could provide feedback to the resource manager.

2.2.3. Intel Threading Building Blocks

Intels Threading Building Blocks (TBB) [Rei07] are a C++ template library, similar to the C++ Standard Template Library (STL) [PLMS00]. TBB provides data structures and algorithms that simplify parallel programming in C++. Instead of directly programming threads, the TBB allow the programmer to specify independently executable work packages (i.e. *tasks*). Tasks are ready for execution once all their input dependencies have been fulfilled. The TBB runtime then schedules these tasks to the cores available to the application. Similar to the Master/Worker-Paradigm (see Section 2.2.1), it enqueues these tasks in *task queues*. For each core available to the application, a task queue is used. Whenever the task queue on one of the cores runs empty, the TBB runtime uses work-stealing techniques [BL94] to transfer work from a random other queue. This avoids using a global task queue, which would limit scalability. Using TBB to express parallelism with tasks allows developers to express more concurrency and finer-grained concurrency than it would be possible with threads, leading to increased scalability [Rei07] i.e., the application can make use of additional cores as long as enough tasks are available. The TBB Frequently Asked Questions [Int14] suggests that the programmer should specify at least ten times the number of tasks than the number of utilized cores to make the work stealing scheduler the most efficient.

"Out of the box", applications written with TBB are moldable applications (see Section 2.2). However, the number of utilized cores (i.e. the number of worker threads) can be adjusted by the application. By slicing the application in multiple *phases* and adapting the number of worker threads per phase the application can be made malleable. In principle, the concept of dedicated work queues per core and work stealing among the cores would allow malleability by spawning additional worker

threads and stealing work from existing queues and/or by terminating worker threads and transferring the remaining tasks to other work queues [SS11]. It is reasonable to assume that future versions of TBB might directly support malleability.

2.2.4. Malleable Software Pipelines

This subsection is based on:

[JPK+13] J. Jahn, S. Pagani, S. Kobbe, J.-J. Chen, and J. Henkel. Optimizations for Configuring and Mapping Software Pipelines in Many Core Systems. In *Design Automation Conference (DAC)*, 2013.

Software pipelines are a well-established means to parallelize stream-processing applications, among which are very common image/video and networking applications [TKA02]. They consist of multiple stages, each processing subsequent iterations on a stream of input data. Each stage i requires the time c_i to compute each iteration. The output data o_i of one stage i forms the input data e_{i+1} of its direct successor. There is no further communication. A malleable software pipeline [JPK+13] can reduce the number of its stages (and thus the number of cores used) at runtime by fusing consecutive stages (see Figure 2.6) so they can be mapped to the same core. Consequently, no communication through the Network-on-Chip between these two stages is necessary. The time required to compute an iteration for the fused stage equals the sum $c_i + c_{i+1}$ of the time required for each stage, however, communication delays between the fused stages vanish. Fused stages can be fused again with other, consecutive stages. Fused stages can be split through fission until the initial degree of parallelism is restored.

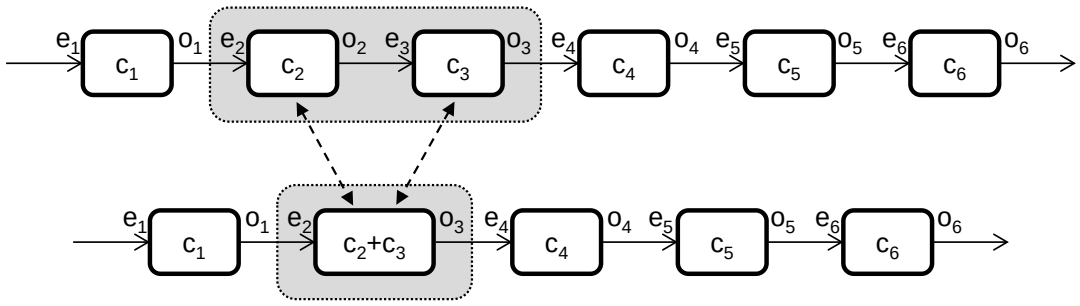

Figure 2.6.: Malleable software pipelines. Adopted from [JPK+13]

The actual *performance* of a malleable software pipeline is determined by its throughput which is limited by the slowest execution time of $e_i + c_i + o_i$ of all stages i. To achieve the best performance, the stages have to be fused differently for different

core allocations. By measuring e_i, c_i, and o_i for each stage, the fusions can be adapted for variations in workload. Additionally, the malleable software pipeline may inform the resource manager about potential performance improvements by allocating additional cores. In [JPK+13], a dynamic programming based solution for optimal fusions of stages has been presented that embodies the capability to provide information to the resource manager. It was shown how the distributed resource management developed in the scope of this thesis can be applied to malleable software pipelines.

2.3. Multi-Agent Systems

As computing systems are continuously getting more complex, it becomes harder to manage them from one central point. Starting with IBM's autonomic computing initiative [Hor01], there has been a clear consent that future computing systems should be self-organizing and self-optimizing to be able to handle the always-growing complexity. This means that the components within a large system have to configure and optimize themselves independently to operate efficiently as a whole.

Multi-Agent Systems [Wei99] are a promising approach to achieve these goals. A Multi-Agent System is composed of multiple interacting intelligent Agents and is typically used to solve problems that are difficult or impossible for an individual Agent or a centralized decision authority to solve. In [JSW98] the major characteristics of Multi-Agent Systems are defined as follows:

- Each Agent has only incomplete (i.e. local) information and is restricted in its capabilities
- System control is distributed
- Computation is asynchronous, i.e. the Agents operate independent of each other (except rather seldom communications)

Agents use communication protocols to exchange information and to decide on how to act. In the following, the Contract Net Protocol (Section 2.3.1) and Gossip Protocols (Section 2.3.2) are presented as they jointly inspired the design of the protocol used by the distributed resource management developed in the scope of this thesis (as presented in Chapter 5).

2.3.1. The Contract Net Protocol

The Contract Net Protocol [Smi80] was developed to specify communication and control for nodes in a distributed problem solver. The Contract Net Protocol or extended versions thereof are used in various Multi-Agent Systems to solve the so-called *connection problem* that describes the situation in which an Agent needs to

find an other suitable Agent to (jointly) solve a problem, e.g. assign a task to the other Agent. The connection problem is solved by negotiations between the participating Agents. According to [Smi80], the protocol has four important properties:

- It does not involve centralized control
- There is a two-way exchange of information
- Each party to the negotiation evaluates the information from its own perspective
- Final agreement is achieved by mutual selection

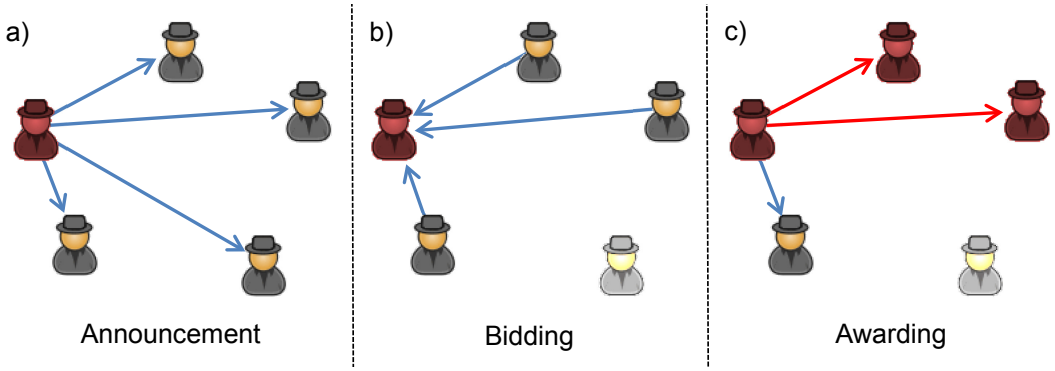

Figure 2.7.: The three core steps (Announcement, Bidding and Awarding) used by the Contract Net Protocol

Being a high-level protocol, it defines in multiple steps *what* the nodes should say to each other, rather than *how* to say it. These steps (visualized in Figure 2.7) are:

Announcement: The Agent that has a problem to be solved announces this problem to all other Agents or a subset thereof. The announcement must contain a specification of the problem. Additionally it might contain constraints that must be fulfilled by the other Agents in order to be eligible to *bid* for the announced problem.

Bidding: Agents that received the announcement decide for themselves whether they have the capabilities to participate in solving the problem. They calculate the *costs* that would occur if they participate. If they decide to participate in solving the problem, they place a bid by sending a message to the announcing Agent that contains information on their capabilities as well as the calculated costs.

Awarding: Once the initiating Agent received enough bids or a timeout occurred, it decides which of the bids (or, which parts of which bids) to accept in order to solve the problem jointly. The result of this process is communicated to all Agents that submitted a bid, even if they have not been selected to participate in solving the problem.

In summary, the Contract Net Protocol specifies a protocol that may be used by an Agent to identify other Agents in order to jointly solve a problem. However, it does not specify how to make any decisions, i.e. it does not formulate any objective functions nor does it specify how to find the final agreement. The distributed resource management developed in the scope of this thesis uses a communication protocol quite similar to the Contract Net Protocol. It bases its decisions on the specified objective function (see Section 5.2) and uses the adaptive application performance modeling presented in Chapter 4 to evaluate the objective function with respect to application mapping.

2.3.2. Gossip Protocols

Gossip Protocols are another typical communication scheme used in Multi-Agent systems. They reach back to the mathematical theory of epidemics described in [Bai57]. Today, they are often employed in distributed systems to enable autonomic self-management, the dissemination of information, to compute aggregates [JMB05], etc.. The most important properties of gossip protocols are [Bir07]:

- Periodic, pairwise interactions between Agents
- Only small amounts of information are exchanged during these interactions
- After the interaction, the state of one or both of the Agents changes in a way that reflects the state of the other Agent
- There is some kind of randomness in the peer selection

An Agent that uses the uses a gossip protocol contains (at least) two components: an active gossiping component and a passive component that reacts on incoming gossip messages. Listing 1 and Listing 2 show the most simple implementation of these components. The periodic interactions typically are referred to as *steps*. Figure 2.8 shows how the random selections of two peers in each steps leads to the dissemination of information initially only known to one Agent after a small number of steps.

Listing 1: Active Gossip

```
while true do
    select (random) peer p;
    send own state s to p;
    sleep until next periodic step;
end
```

Listing 2: Passive Gossip

```
while true do
    wait for state s_p from peer p;
    own state s ← merge s with s_p;
end
```

The goal that shall be achieved by gossiping is defined by the **merge** function that merges the own state of an Agent with the new information it just received.

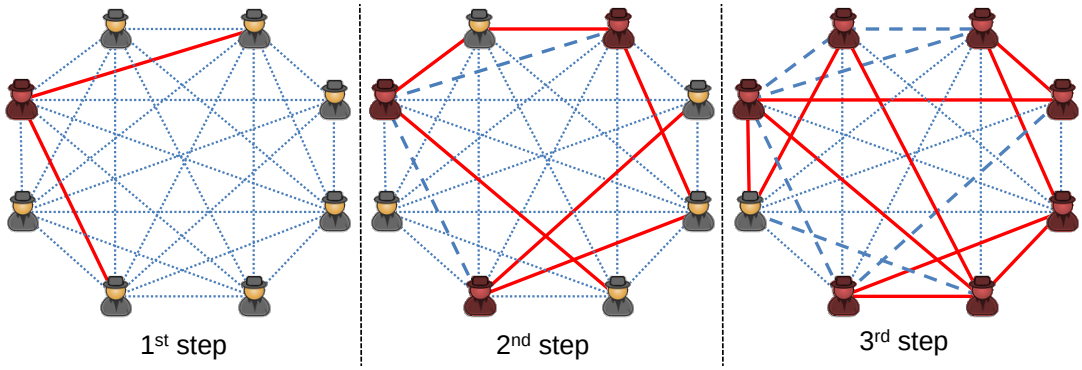

Figure 2.8.: Information dissemination by using a gossip protocol (two random peers selected in each step)

For instance [RNV09] uses a gossip protocol to minimize a global objective function which is the sum of the local objective functions of the different Agents. In each step, each Agent averages its state with a randomly chosen neighbor and adjusts the average using the gradient of its local function that is computed with stochastic errors. The paper proves that for convex local objective functions the global state converges to an optimal solution almost surely.

The random selection of peers allows gossip protocols to operate efficiently without centralized infrastructure and to disseminate information throughout the system. However, in many settings, new information generated at individual Agents is most interesting to Agents that are spatially nearby. Spatial gossip [KKD04] proposes to vary the probability to select a peer with the spatial distance, i.e. nearby Agents will receive the information sooner, and on average the communication distances are reduced.

2.4. Operating Systems for Many-Cores

Traditional operating systems like the Linux kernel [Fou] can manage the resources of a many-core system if it has shared memory and cache coherency. In [BWCM+10] it has been shown that the Linux kernel does scale well up to 48 cores in an AMD multi-socket system. In theory (and praxis) the Linux kernel supports up to 4096 cores. The SGI UV System [Sil14], a huge NUMA[3] system that takes a whole server rack, scales to 4096 threads (2048 cores on 256 CPU sockets) and up to 64TB of cache-coherent, global shared memory in a single system which runs a off the shelf Linux operating system. However, there is no *smart* application mapping – to achieve good performance on such huge systems, a mapping of tasks to cores has to be done

[3]Non-uniform memory access, the memory access time depends on the memory location relative to the processor, i.e. a processor can access its own local memory faster than non-local memory.

before application execution. For systems without cache coherency, novel approaches and operating system designs are required. In the following, two research operating systems tailored for (on-chip) many-core systems are presented.

2.4.1. Tessellation Operating System

The fundamental component of the Tessellation Operating System [tes14] is a resource-guaranteed *cell* which provides guaranteed fractions of system resources (such as processor cores, cache, network or memory bandwidth, fractions of system services, etc.). The software running within each cell has full user-level control of the resources assigned to the cell, including CPU cores and memory pages [CEH+13]. Applications may consist of multiple cells that are connected by secure channels. Also the operating system services execute in their own separated cells. The Tessellation OS derives its name from the space-time partitioning approach that tessellates cells across the space of hardware resources.

Resources are virtualized using space-time partitioning, a multiplexing technique that divides the hardware into a sequence of simultaneously-resident spatial partitions [LKB+09]. With space-time partitioning, cores and other resources are gang-scheduled [Ous82]. The individual cells thus provide an environment that behaves similar to a dedicated machine. Each application uses an user-space scheduler to schedule the applications tasks to the cores in the cell. Tessellation separates the global decisions about the allocation of resources to cells from application-specific scheduling of resources within cells.

An adaptive resource redistribution among cells occurs at a coarse time scale to amortize the decision-making cost. A centralized *Resource Allocation Broker* collects periodic performance reports from each application and the hardware components to make its decisions [CBC+10]. The Resource Allocation Broker is generic and allows the implementation of different policies, e.g. a reactive control that assigns additional resources until a specified performance goal is achieved. Other, more advanced policies could be implemented, however.

2.4.2. Barrelfish Operating System

The research operating system Barrelfish [Bar14] explores how to structure an operating system for future multi- and many-core systems. It is motivated by the rapidly growing number of cores per chip, which leads to a scalability challenge, and heterogeneous hardware resources.

Barrelfish follows the principles of a so-called multi-kernel operating system [BBD+09]. It instantiates one lightweight kernel on each core and treats the whole machine as a network of independent cores, assumes no inter-core sharing at the

lowest level, and moves traditional operating system functionality to a distributed system of processes that communicate by means of message-passing – even on systems with shared memory and/or cache coherency. In contrast to traditional operating system that share state among cores, Barrelfish follows the "shared nothing" design model. This enforces an event-driven control flow which however already is the common programming style of operating systems. Barrelfish divides the code running on each core into a privileged-mode *CPU driver* which operates purely local on the core, and a distinguished user-mode *monitor* process that handles all inter-core coordination. The rest of Barrelfish consists of device drivers and system services, which run in user-level processes as in a microkernel [Lie95]. The message-passing based "shared nothing" design model allowed to port Barrelfish efficiently to the Intel SCC many-core platform [PRB10] and various other platforms.

All cores allocated to an application are referred to as *application domain*. Cores in Barrelfish might be time-multiplexed among different applications, i.e. there is a spatial and temporal multiplexing of applications. Each application runs a set of dispatchers, one per core in the application domain, which collectively implement an application-specific user-level thread scheduler [PSB+10]. Similar to the Tessellation operating system, the dispatchers on different cores of on application are gang scheduled by the concept of *phase-locked schedulers*.

The placement of applications onto cores takes place in the *long-term* scheduling, taking into account application requirements, system load, and hardware details. A centralized placement controller uses information on the applications resource demands along with knowledge of the momentary hardware utilization to determine a suitable set of hardware resources for the applications. System state and application requirements are stored as first-order logic expressions [AW+07]. The placement policy is then implemented in Prolog [PSB+10]. Currently, a load balancing policy to distribute application load over all available cores is implemented [Pet12].

2.5. Runtime-adaptive Systems at CES

Runtime-adaptive systems of different domains are and always have been one of the primary research directions at the Chair for Embedded Systems (CES). The broad scope of research covers runtime-adaptive network-on-chip architectures [FAEH07, FAEH08], runtime-adaptive instruction set architectures [BSKH07, BSKH08], runtime-adaptive task-management [AFKH08, JAFH11, JPK+13], thermal management [EFH09], memory management [HBH12], sensor networks [SBHH15], and others like research with respect to reliability [HBB+11]. This Section briefly introduces the topics related to the runtime-adaptive resource management presented in this thesis.

ADAM [AFKH08] was the first approach for runtime resource management using a Multi-Agent-System developed at the Chair for Embedded Systems. It uses an

hierarchy of Agents, where each Agent is responsible for resource management of a subset of the cores (referred to as *virtual cluster*) in the system. If an Agent is not able to answer a resource request within its virtual cluster, it forwards the request to a centralized global control Agent that then might choose an other Agent or triggers a re-clustering of the resource subsets managed by each Agent. However, this global control Agent was left for future work and therefore cannot be compared to the resource management developed in the scope of this thesis.

TAPE [EFH09] (thermal-aware agent-based power economy) is a distributed thermal management scheme for on-chip many-core systems. It also employs the concept of a fully distributed agent-based system in order to deal with the complexity of thermal management in many-core systems. However, in TAPE each core is assigned its own agent which is able to negotiate with its immediate neighbors (i.e. adjacent cores). Thermal management itself is performed by distributing power budgets which dictate task execution among the cores. Thus the agent negotiation consists of distributing this power budget based on the concept of supply-and-demand, taking the currently measured temperatures into account. Since each agent is only able to trade with its neighbors, multiple agent negotiations are required to propagate power budget across the chip. At start-up, the available tasks are randomly mapped on the cores in the grid and then continuously remapped when either there is no power budget available in a node or the difference of temperatures in the neighboring nodes goes beyond certain threshold [EFH09]. TAPE is highly related to the distributed resource management presented in this thesis as both approaches share the concept of a fully distributed Multi-Agent-System, however aiming at a different optimization goal. In contrast to TAPE, which associates an Agent with each *core*, the work presented in this thesis associates an Agent with each *application*. TAPE assumes applications to consist of tasks that are executed on single cores and that may freely be re-allocated to different cores, whereas the resource management presented in this thesis uses a different application model in which each application consumes multiple cores concurrently.

The same optimization goal as the one used in this thesis was persuaded in [JPK+13]. The concept of malleable software pipelines is used as application model. However, the problem is simplified such that it only considers the *number* of cores to allocate to each application but not their topological location on the chip. The resulting optimization problem is solved using dynamic programming. The reduced complexity of the optimization problem allows to solve it optimally centralized for many-core systems with a reduced number of cores or in a distributed hierarchical manner for large many-core systems, i.e. it does not use a Multi-Agent-System as the approaches mentioned before. An improved version of the optimization problem that also considers the available memory bandwidth of the memory controllers is presented in [JPCH13].

The main idea of *invasive computing* [THH+11] is to introduce resource-aware programming support in the sense that a given program gets the ability to explore

and dynamically spread its computations to processors similar to a phase of invasion, then to execute portions of code of high parallelism degree in parallel based on the available (invasible) region on a given multi-processor architecture. Afterwards, once the program terminates or if the degree of parallelism should be lower again, the program may enter a retreat phase, deallocate resources and resume execution again, for example, sequentially on a single processor. To support this idea of self-adaptive and resource-aware programming new programming concepts, languages, compilers and operating systems are necessary as well as architectural changes in the design of MPSoCs (Multi-Processor Systems-on-a-Chip) to efficiently support invasion, infection and retreat operations by involving concepts for dynamic processor, interconnect and memory reconfiguration [HHB+12].

As systems with 1000 or more processors on a single chip are expected in the year 2020 [ITR13], static and central management concepts to control the execution of all resources might have met their limits long before and are therefore not appropriate. Invasion might provide the required self-organising behaviour to conventional programs for being able to provide scalability, higher resource utilisation, required fault tolerance, and of course also performance gains by adjusting the amount of allocated resources to the temporal needs of a running application. The resource management developed in the scope of this thesis is part of the invasive computing project and enables and exploits these properties of invasive applications.

3. Models and Assumptions

Inspired by trends of on-chip many-core research, as shown in Chapter 2, the resource management presented in this thesis is tailored to the following assumptions:

- The on-chip many-core system has significantly more cores N than concurrently running applications M. A Network-on-Chip connects the cores, the primary means of communication is message passing. The many-core system is described by the set $\mathcal{C}$ of all cores c_i, i.e. $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$. Each core c_i has a unique topological location on the chip.

- Applications $\{A_1, A_2, \ldots, A_M\}$ consist of multiple phases (e.g. the computation kernels EV, HH, and BT from the motivational example NWChem in Section 1.2.1). Each phase P_j consists of multiple smaller tasks (e.g. parallel implementations of a computation kernel, see Section 3.2). Each phase of each application may execute on multiple cores concurrently, however, there is always only one active phase per application.

- Applications are assumed to be malleable (see Section 2.2), i.e. they are able to adapt their degree of parallelism to the allocated cores. Applications can utilize additional cores at any point in time, as long as there are additional tasks ready for computation.

- Applications are assumed to release cores when required by the resource manager. Depending on the implementation of the malleability of the application, this release may take place immediately (tasks are migrated [JAFH11] or are restarted on other cores), or if the application consists of short-running tasks directly after the tasks currently assigned to the respective cores have finished execution.

- Resource management means to (re-)allocate a subset of cores $C_{A_i} \subseteq \mathcal{C}$ to each application A_i such that no core is assigned to two applications at the same time, i.e. for all A_i, A_j with $i \neq j$, $C_{A_i} \cap C_{A_j} = \emptyset$. Cores are exclusively allocated to applications at certain points in time, i.e. there is no need for multi-tasking per core. Current many-core operating system research [CEH+13, OSK+11, SATG+07] emphasizes the advantages of temporally exclusive allocation of cores to applications to avoid multiplexing overhead. The goal of runtime resource management is to dynamically select these subsets C_{A_i} in a way that maximizes a predefined optimization goal (see Section 5.2).

3.1. Many-Core Platform Model

In line with the definition given in [VBC11], the targeted architecture of this work is basically is the greatest common divisor of the many-core architectures presented in Section 2.1: a homogeneous on-chip many-core system as depicted in Figure 3.1. Each core is capable to execute any application and system service, i.e. there are no specialized cores. Cores are connected through a Network-on-Chip (NoC), communication is implemented via message passing. The topological location of each core c_i is given by its x- and y-coordinates (c_i^x, c_i^y) in the 2d-mesh. This also means, the performance of an application depends not only on the number of allocated cores but also on the mapping of the applications tasks to cores and the (relative) topological location of these cores on the chip.

Please note: The protocols and mechanisms developed and presented in Chapter 5 are not limited to homogeneous NoC-based many-core systems. However, the application performance estimation presented in Chapter 4, and the exploitation of spatial neighborhood are tailored to this platform model.

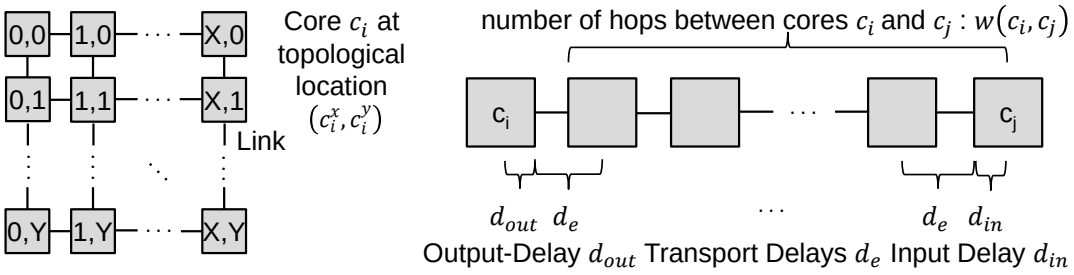

Figure 3.1.: System and Network Delay Model Components as described in [LK03]

$$\begin{aligned} w(c_i, c_j) &= (|c_i^x - c_j^x| + |c_i^y - c_j^y|) \\ d_{NoC}^{idle}(s, c_i, c_j) &= d_{out} + d_e \cdot (s + w(c_i, c_j)) + d_{in} \end{aligned} \tag{3.1}$$

The NoC communication delay model based on xy-routing [LK03] is used. In an idle NoC (i.e. no communication is taking place), the communication delay d_{NoC}^{idle} for a message of size s (see Equation (3.1)) between two cores c_i and c_j consists of three components: The first component d_{out} is the time spent at c_i to output the message to the NoC. The second component is the time d_e the message takes to travel through the NoC (the delay that each link induces to process one byte) multiplied by the sum of s and the number of needed hops $w(c_i, c_j)$ to send the message between the cores c_i, c_j. The third component is the time d_{in} that core c_j spends for processing the incoming message. However, this delay is only valid in an idle system. In [LK03, YXSP10] it has been argued that on average the link contention and buffer utilization in the routers lead to a linear relation between

the experienced delay d_{NoC} of a message and the message size s multiplied by the required number of hops $w(c_i, c_j)$ as shown in Equation (3.2).

$$d_{NoC}(s, c_i, c_j) \approx d_e \cdot s \cdot w(c_i, c_j) \tag{3.2}$$

3.2. Application Model

The *speedup* $S_{A_i}(C_{A_i})$ always refers to the relative execution time $T_{A_i}(C_{A_i})$ of an application A_i compared to being executed on only one core ($T^{\#}_{A_i}(1)$), as shown in Equation (3.3). The *efficiency* $E_{A_i}(C_{A_i})$ of application execution is defined as the achieved speedup per core, see Equation (3.4). The number of cores in C_{A_i} is given by $|C_{A_i}|$. The notion of $S_{A_i}(C_{A_i})$, $T_{A_i}(C_{A_i})$, and $E_{A_i}(C_{A_i})$ refers to the speedup, execution time, and efficiency of A_i when considering the actual mapping of the cores in C_{A_i}, whereas $S^{\#}_{A_i}(k)$, $T^{\#}_{A_i}(k)$, and $E^{\#}_{A_i}(k)$ refer to the average speedup, execution time, and efficiency of A_i when only considering the number of cores $k = |C_{A_i}|$ in C_{A_i}.

$$\begin{aligned} S_{A_i}(C_{A_i}) &= \frac{T^{\#}_{A_i}(1)}{T_{A_i}(C_{A_i})} \\ S^{\#}_{A_i}(k) &= \frac{T^{\#}_{A_i}(1)}{T^{\#}_{A_i}(k)} \end{aligned} \tag{3.3}$$

$$\begin{aligned} E_{A_i}(C_{A_i}) &= \frac{S_{A_i}(C_{A_i})}{|C_{A_i}|} \\ E^{\#}_{A_i}(k) &= \frac{S^{\#}_{A_i}(k)}{k} \end{aligned} \tag{3.4}$$

The *relative progress* p_{A_i} quantifies how much work of the application A_i has been completed. A progress of $p_{A_i} = 0$ means that the application has not done any computation while $p_{A_i} = 1$ means that the application has finished all of its work. If p_{A_i} is given, then the application requires time $T^{finish}_{A_i}(C_{A_i})$ (see Equation (3.5)) to finish execution on C_{A_i}.

$$T^{finish}_{A_i}(C_{A_i}) = \frac{(1 - p_{A_i}) \cdot T^{\#}_{A_i}(1)}{S_{A_i}(C_{A_i})} = (1 - p_{A_i}) \cdot T_{A_i}(C_{A_i}) \tag{3.5}$$

Applications consist of computation *phases* P_j, i.e. application $A_i = \{P_1, P_2, \ldots P_J\}$. The phases, e.g. computation kernels, are executed in order (P_{j+1} is executed after P_j), however repetitions of phases or groups of phases are allowed. Each phase P_j has characteristic properties. Figure 3.2 shows the components of the application

model. For instance, phases P_2 to P_4 might be the three computational kernels performing the linear algebra calculation [KAB+00] from the motivational example in Section 1.2.1.

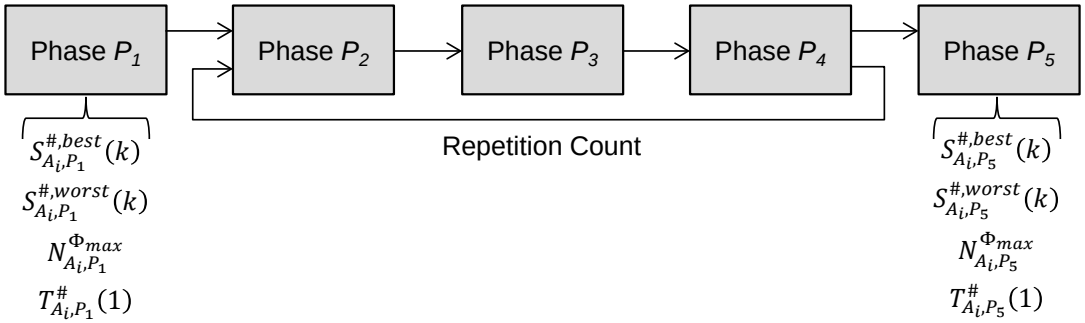

Figure 3.2.: Modeled application A_i consisting of five characteristic phases $\{P_1, \dots P_5\}$ with a repetition of phases P_2 to P_4

The parameters of each phase P_j are:

- The speedup functions $S^{\#,best}_{A_i,P_j}(k)$ and $S^{\#,worst}_{A_i,P_j}(k)$ describe the speedup of each phase P_j of application A_i when using $k = |C_{A_i}|$ cores in the best-case and in the worst-case. Typically, the best-case is achieved when the cores in C_{A_i} are spatially close to each other, whereas the worst-case occurs when the cores are spatially distant from each other (see Section 4.3.2). Chapter 4 details how the achievable speedup $S_{A_i,P_j}(C_{A_i})$ is estimated from $S^{\#,best}_{A_i,P_j}(k)$ and $S^{\#,worst}_{A_i,P_j}(k)$.
- The execution time of P_j when executed on only one core is given by $T^{\#}_{A_i,P_j}(1)$.
- The best-suited number of cores $N^{\Phi_{max}}_{A_i,P_j}$ for application A_i in phase P_j is determined by choosing $k = |C_{A_i}|$ such that the product $\Phi(k)_{A_i,P_j}$ (see Equation (3.6)) of the speedup $S^{\#,best}_{A_i,P_j}(k)$ and the efficiency $E^{\#}_{A_i,P_j}(k) = S^{\#,best}_{A_i,P_j}(k)/k$ (i.e. the speedup improvement per core) of each application phase P_j is maximized, as proposed by [Dow98, EZL89].

While executing phase P_j of application A_i on more than $N^{\Phi_{max}}_{A_i,P_j}$ cores may yield a higher speedup, the efficiency typically degrades significantly, for instance as shown in Figure 3.3. In Section 5.4.6, this knowledge with respect to $N^{\Phi_{max}}_{A_i,P_j}$ is exploited to select the resource management strategy to be used to optimize the set of cores C_{A_i,P_j} allocated to application A_i in phase P_j.

$$\Phi(k)_{A_i,P_j} = S^{\#,best}_{A_i,P_j}(k) \cdot E^{\#}_{A_i,P_j}(k) = \frac{S^{\#,best}_{A_i,P_j}(k)^2}{k} \tag{3.6}$$

The application knowledge is obtained by offline analysis. This analysis needs to be performed for each phase P_j of each application A_i that should be executed on the system.

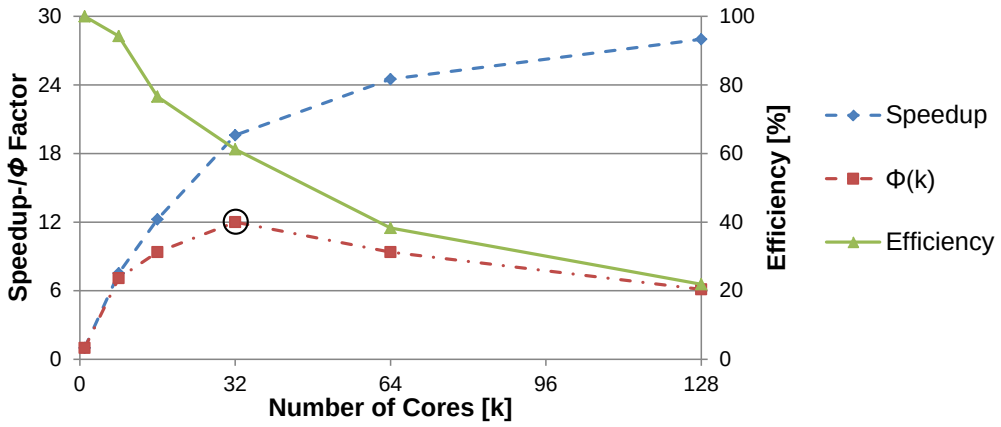

Figure 3.3.: The relationship of Speedup $S^{best}_{A_i,P_j}(k)$, Efficiency $E^{\#}_{A_i,P_j}(k)$, and $\Phi(k)$ of a computation kernel (P_j) when executed on different numbers $k = |C_{A_i}|$ of cores (Speedup values taken from [KAB+00])

If the phases of the application are not known, the whole application may be represented by a single phase. Alternatively, the phases may be determined in the analysis process [KKH13]. For instance, Figure 3.4 shows the parallel profile of a task-graph generated with the Task Graphs For Free (TGFF) tool [DRW98] and the identified phases in that task-graph. The task scheduler for multi-core systems presented in [THW02] is used to split the task-graph into a linear sequence of *steps*, where each step consists of the tasks that are ready for execution once all tasks in the previous steps have completed execution, i.e. all tasks of one step may be executed in parallel. Successive steps with a similar degree of achievable parallelism are grouped into the same phase. The speedup curves for each phase are then individually determined. If the computational kernels of an application are known (e.g. as in the motivational example in Section 1.2.1), the kernels represent suitable phases and can then directly be profiled.

Techniques for intra-application scheduling and load balancing are not within the scope of this thesis. For instance Runtime Task-Mapping like the one presented for Adaptive MPI [HLK04], Intel Threading Building Blocks [Phe08], or an online task graph scheduler [CCK12] can be used to map the applications tasks to the cores C_{A_i} that have been allocated to A_i by the resource manager.

Please note: To improve the readability of this thesis, the phase is not always explicitly mentioned. For instance $S^{\#}_{A_i}(k)$ might be used instead of $S^{\#}_{A_i,P_j}(k)$. In theses cases $S^{\#}_{A_i}(k)$ refers to the momentary *active* phase P_j.

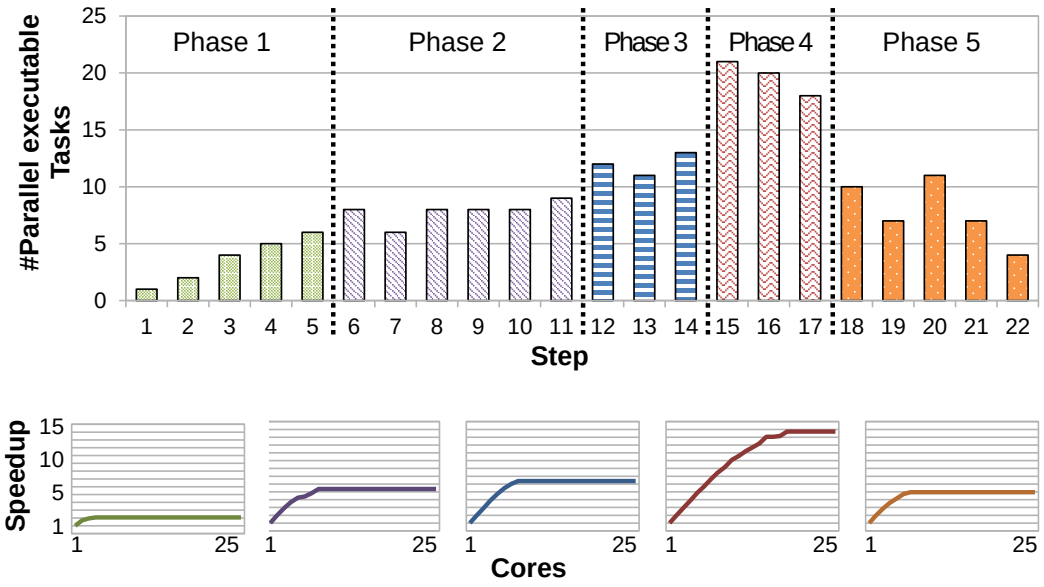

Figure 3.4.: Parallel profile of an application, the identified phases $\{P_1, P_2, \ldots P_5\}$ (consisting of multiple sets of tasks with similar resource demands), and the respective speedup curves $S^{\#,best}_{A_i,P_j}(k)$ [KKH13]

3.3. Operating System Model

An operating system runs directly on top of the hardware and provides an abstract machine interface to the applications. It encapsulates system services that ease using the hardware. In the scope of this thesis, resource management is considered as a part of the operating system and responsible to decide which applications should execute on which sets of cores at any point in time. Applications use the interface provided by the operating system to provide information on their resource demands to the resource manager which in turn allocates resources to the application.

At least the following functionality must be provided to realize the resource management developed in the scope of this thesis:

- The operating system must be able to constrain each application A_i to execute only on the set of allocated cores C_{A_i}.
- The operating system must provide a mechanism to communicate between different cores and software instances. This is required for the applications to inform the resource manager about updated resource demands, and in the case of distributed resource management, to realize the resource management itself.

These functionalities are available in the research operating systems presented in Section 2.4, as well as in the OctoPOS [OSK+11] operating system, for which

a reduced variant of the presented resource management was implemented and integrated.

Similar to the Barrelfish OS, OctoPOS instantiates one kernel per core[1] and uses message passing between the individual instances. OctoPOS provides a sophisticated Remote Procedure Call (RPC) mechanism to facilitate message passing. Resource management decisions are made on the same cores that are used by the applications. This means, that messages addressed to the resource manager have to be handled on cores that potentially are executing application code that must be interrupted. However, this does not entail preemptive multi-tasking – OctoPOS allows to execute RPCs in a software interrupt context, i.e. interleaved with the actual application.

In the Tessellation OS, the concept of temporarily exclusive allocated resources is called a *cell*, in the Barrelfish OS, this is a *domain* and in the OctoPOS, this is a *claim*. In the remainder of this thesis, C_{A_i} is used to refer to the resources allocated to application A_i.

[1] Actually, there is one instance per cache coherent *tile*. However, following the system model presented earlier, there is only one core per tile.

4. Adaptive On-the-Fly Application Performance Modeling

This chapter is based on:

[KBH15] Kobbe S., L. Bauer, and J. Henkel. Adaptive on-the-fly application performance modeling for many cores. In *IEEE/ACM 18th Design Automation and Test in Europe Conference (DATE´15)*, 2015.

Many-core systems pose the challenge of efficient system utilization, especially if application properties and the set of concurrently executing applications rapidly vary at runtime. Therefore, when running multiple applications in parallel, it is essential to allocate the *right* set of cores to the *right* applications at runtime and thus to achieve a high degree of efficiency. This not only entails the number of cores but also their topological location on the chip. For example, the work in [MBS+05] has shown a 42% reduction of application execution time through sophisticated mapping.

4.1. Problem Description

To be able to select the *right* cores dynamically, accurate performance estimates are necessary. These estimates rely on so-called application performance models. The performance of an application depends on the system on which it is executed, its input data, and the system load caused by concurrently running applications. As the number and type of concurrently executing applications is not known a priori, estimations that are solely based on offline analysis will most likely result in inaccurate estimates and thus in disproportionate mappings. A high number of estimates is calculated at runtime to find a suitable allocation of cores to each application – whenever a change in the workload triggers a new application mapping! This requires the performance estimation to have low computational effort. Otherwise, the resulting latency of runtime resource management hampers dynamic changes in the workload of the system. Given a (potential) resource allocation C for application A_i, an application performance model estimates the speedup $S_{A_i}(C)$ application A_i achieves. The estimated speedup $S_{A_i}(C)$ depends on characteristic properties of A_i as well as C.

Please note: In this thesis, the speedup $S_{A_i}(C)$ of an application A_i is used as a relative performance metric, i.e. the terms "application performance" and "application speedup" are used interchangeably.

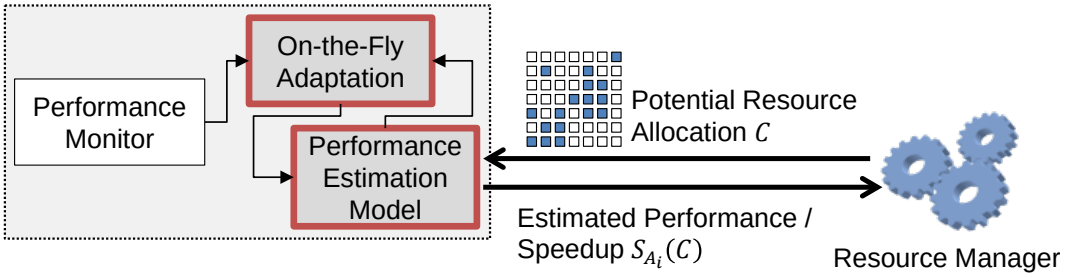

Figure 4.1.: Principal components and interactions of the resource manager and the adaptive resource-aware application performance model

Sometimes, it is possible to base these estimates solely on static application knowledge determined by offline analysis. However, the performance of an application depends on the actual system it is executed on [HZQ+13, LNC13], its input data [PSK+12], and the system load caused by concurrently running applications. As the number and kind of concurrently executing applications is not known a priori, estimations solely based on offline analysis will most likely result in inaccurate estimates and thus in disproportionate application mappings. Therefore the achieved performance of the application is measured. The speedup model then has to adapt its parameters to minimize the difference of measured and estimated speedups. Figure 4.1 shows the principal components and interactions of the resource manager and the adaptive resource-aware application performance model.

4.2. Related Work

The importance of application performance models for many-core resource management ranges from on-chip many-core systems [ATBS13, KBL+11] over high-performance computing [SLS07], to cloud computing and grid computing environments [DA06, KKG+11]. Here, the models are used to decide on the number of cores that are allocated to each application.

Relevant related work is on application performance models for parallel applications [HZQ+13, Dow98], (online) learning of application performance estimation [IDSSM05, BIM08, HMS+11], and resource management for many-core systems that use speedup models (with and without adaptation) [HMS+11, ATBS13, SLS07, ZA10, YGSP13] which will be addressed in Section 5.1. In the following sections, performance models for parallel applications and online-learning approaches are presented.

4.2.1. Extended Amdahl's Law for on-Chip Interconnect

Both Amdahl's law [Amd67] that describes the theoretical maximal speedup an application can achieve based on its sequential code segments, and Gustafson's law [Gus88] that describes how much the problem size could be increased when using multiple cores do not explicitly consider the on-chip interconnect of todays and future many-core systems – both naturally modeled an application as a pure computational workload without explicit considerations on inter-core communication. When more parallelism is exploited by mapping tasks onto more cores, inter-core communication overhead may rise rapidly. The authors of [HZQ+13] propose an updated version of both laws; however, they do not consider the (absolute or relative) topological location of the cores.

Amdahl's law allows to calculate the achievable speedup of an parallel application when executing on k cores. It assumes that a fraction f of the application is parallelizable while the reminder of the application consists of sequential code that can not be speedup by more than one core. Equation (4.1) shows the resulting speedup function.

$$S(f,k) = \frac{1}{(1-f) + \frac{f}{k}} \tag{4.1}$$

To also include the communication of the application, in [HZQ+13] f it is split into four fractions (see Equation (4.2)): f_c^s and f_c^p, respectively, express the sequential and parallel fractions of the application that are spent for computation (i.e. the parameters of Amdahl's law) while f_t^s and f_t^p express the (sequential and parallel) fractions of the application that is used for transmission of data. $\hat{f}$ is used to express the tuple of application parameters $(f_c^s, f_c^p, f_t^s, f_t^p)$.

$$f_c^s + f_c^p + f_t^s + f_t^p = 1 \tag{4.2}$$

Additionally to the improved description of the application properties, they included the parameter i, the number of interconnects, into the speedup function that describes the hardware the application is executed on. It can be explained as the number of links of a single core in the NoC of many-core processors. Equation (4.3) shows the resulting speedup function which is a two-dimensional extension of Equation (4.1) [HZQ+13].

$$S\left(\hat{f},k,i\right) = \frac{1}{f_t^s + f_c^s + \frac{f_c^p}{k} + \frac{f_t^p}{i}} \tag{4.3}$$

While the extended model explicitly considers the fractions of sequential and parallel computation and communication $\hat{f}$ (i.e. properties of the application) and

properties of the hardware i, the topological location of the cores allocated to the application is not considered. This means, that the model is not able to differentiate between a *good* set for cores C and a *bad* one.

4.2.2. Downey's application performance model

A generic application performance model for parallel workloads has been presented in [Dow98]. It models achievable speedup of real-world applications when executed on computer clusters very well. The model requires two parameters: the average parallelism P of an application and its variance in parallelism σ. Equation (4.4), adapted from [Dow98], shows how for a given $k = |C|$ the speedup $S^{\#}(k)$ of an application is calculated if these parameters are given. High variances in parallelism σ require a different calculation of the speedup (Equation (4.4b)) than applications with little or no variance at all (Equation (4.4a)). The model has been validated against various benchmark applications on shared- and distributed memory systems. The parameters of the model correspond to measurable program characteristics. This insight into the values and distributions of these parameters in a real workload allowed to implement a workload generator [Fei05] which allows to generate application profiles and workload scenarios similar to those found in parallel high-performance computing clusters. This generated workload profiles are widely used to evaluate many-core resource management [Fei03, SLS07, SCH11, ATBS13].

$\sigma < 1$ (low variance):

$$S^{\#}(k) = \begin{cases} \frac{kP}{P + \frac{\sigma(k-1)}{2}} & \text{if } 1 \leq k \leq P \\ \frac{kP}{\sigma(\frac{P}{2}) + k(1-\frac{\sigma}{2})} & \text{if } P \leq k \leq 2P-1 \\ P & \text{if } k \geq 2P-1 \end{cases} \tag{4.4a}$$

$\sigma \geq 1$ (high variance):

$$S^{\#}(k) = \begin{cases} \frac{kP(\sigma+1)}{\sigma(k+P-1)+P} & \text{if } 1 \leq k \leq P + P\sigma - \sigma \\ P & \text{if } k \geq P + P\sigma - \sigma \end{cases} \tag{4.4b}$$

Similar to the (extended) speedup function following Amdahl's law, the model does not consider the topological location of allocated cores in a many-core system, i.e. it is topology-agnostic and not resource-aware. However, because of its easy parameterability based on offline profiling and the good match to of the estimated speedups to the real values, the speedup model developed in this thesis [KBH15] is based on Downey's application performance model [Dow98].

4.2.3. Machine Learning for Performance Prediction

Due to subtle interactions between architecture and software it is not always possible to accurately model and predict performance for large-scale applications with an analytic model. Other approaches to estimate the achievable performance of an application use machine-learning techniques like artificial neuronal networks [IDSSM05]. Similar to offline-parameterized analytical speedup models, these approaches require an exhaustive training phase before they are able to predict the achievable application performance. Even the authors claim that a neuronal network is not yet generally useful in the absence of an existing database, as the time required to gather each data point in the training set is rather large.

The so far presented application performance models are parameterized offline, i.e. they can not react to changes in e.g. the input data of an application nor to changes in the overall workload of the system, i.e. the influence other applications have on the achievable performance. In [BIM08], artificial neuronal networks were implemented in hardware and continuously trained with the performance metrics of various shared resources in a quad-core processor. They assumed one application to execute on each core to measure the achieved performance. Based on the predictions performance provided by the neuronal networks, a centralized controller decides, which fraction of shared resources to assign to which core.

In contrast to [IDSSM05], the neuronal networks were adapted at runtime by using error backpropagation learning [RHW88]. While the goal of the resource management based on artificial neuronal networks was somewhat different to resource management in a many-core system, it is reasonable to assume that the approach might be adoptable to larger systems. However, the amount of additionally required hardware is considerably high. For the quad-core system running four applications, 16 artificial neuronal networks – each consisting of 52 hardware multipliers – were required.

4.2.4. The SElf-awarE Computing (SEEC) model

SEEC [HMS+11] is a combination of an adaptive speedup model and a resource manager. It follows the idea of autonomic computing [Hor01, ABD+03, KC03], where the system adapts itself to the momentary situation by means of an *observe-decide-act* loop, as shown in Figure 4.2. The SEEC runtime requires the specification of all possible *actions* (e.g., allocate n cores to application A_i) that can affect application performance, each with a predefined estimated impact on system performance. At runtime, it selects a combination of actions for each application in a way that fulfills the goals specified by the application while maximizing the estimated performance using a centralized control approach [MHS+10].

SEEC is capable to adapt to changes in both application and system models by

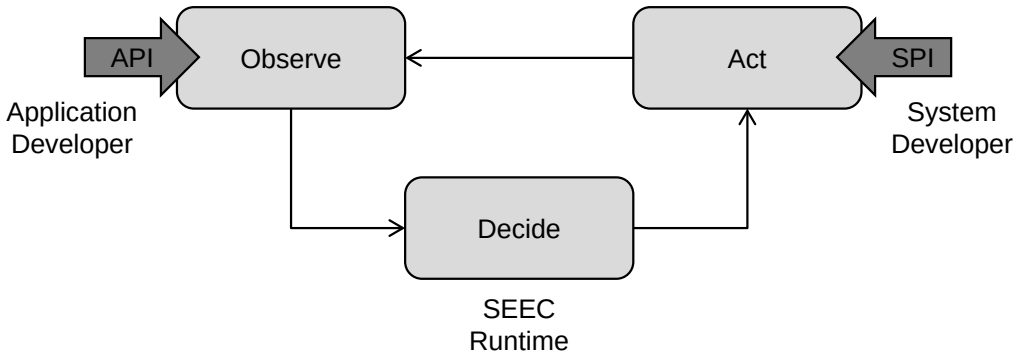

Figure 4.2.: Observe-Decide-Act Loop used by SEEC. The Application Developer uses an application programming interface (API) to report the performance of the application and to specify the achievable goals. The System Developer uses the system programming interface (SPI) to specify possible system configuration actions. The SEEC runtime selects the best combination of actions to achieve the specified goals.

correcting the estimated impact of the different actions on the system performance. After making a decision, it uses the Application Heartbeats framework [HES+10] to monitor the performance of applications and to update the estimated impact of the action with the measured values.

By defining all possible actions the sheer number of possible actions is too large in systems with hundreds or even thousands of cores – especially when not only the number of cores but also their topological location would be considered. Originally, SEEC is tailored to shared-memory systems and it only considers the number of cores. The centralized design of SEEC might impose scalability issues, see Chapter 5.

4.2.5. Summary of Related Work

In summary, there are application performance models that adapt at runtime, consider the topological properties of the cores allocated to an application, or allow on-the-fly performance estimates. However, there is no adaptive model suitable for on-the-fly application performance estimation on NoC-based many-core systems considering the topology of cores.

Therefore, this thesis proposes an adaptive on-the-fly application performance model that considers the topological properties of the cores allocated to an application in NoC based many-core systems. It uses a simple metric of C that can easily be determined to estimate the achievable speedup based on the lower bound and the upper-bound speedup of the application. To handle highly dynamic behavior of workloads not known a priori, the application performance model is continuously adapted at runtime.

4.3. Empirical Analysis of Application Performance

To motivate the resource-aware application performance model developed in this thesis, first some empirical analysis is presented. This analysis is based on a refined application model which allows a fast design space exploration (Section 4.3.1). The resource-awareness is achieved by considering additional properties of C than just the number of cores $k = |C_{A_i}|$. Section 4.3.2 motivates why the *average distance* between cores in C is a suitable choice.

4.3.1. Refined Application Model

Without the loss of generality, the phases of the applications (see Section 3.2) are represented as task-graphs where the tasks are represented as the nodes n_x in a directed acyclic graph. This is shown in Figure 4.3a where an inner task-graph is encapsulated in a larger *block* (flanked by a *begin* node and an *end* node) that is repeatedly executed in an outer loop.

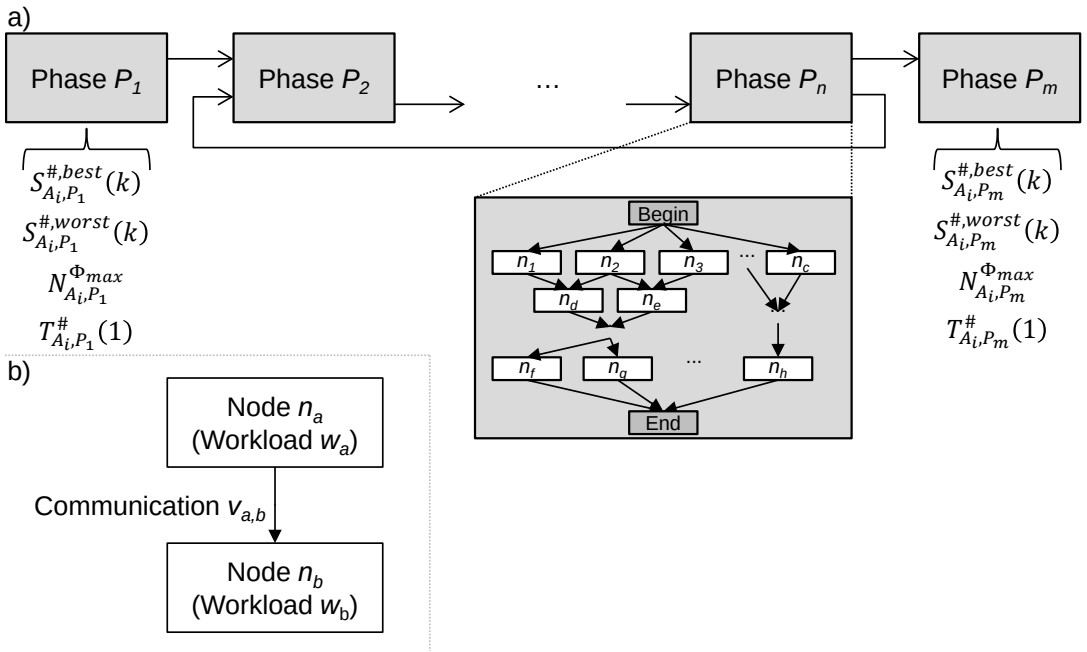

Figure 4.3.: Refined application model which allows a fast design space exploration: Application phases are represented by task graphs.

The nodes n_x correspond to actual computation (expressed as workload w_x for each node n_x) that the application performs and the edges correspond to the communication volume $v_{a,b}$ between two nodes n_a and n_b (see Figure 4.3b). The execution time $T^{\#}_{A_i,P_j}(1)$ of each phase P_j on only one core is the sum of the workload w_x of all nodes $n_x \in P_j$, as shown in Equation (4.5).

$$T^{\#}_{A_i,P_j}(1) = \sum_{n_x \in P_j} w_x \tag{4.5}$$

A node is ready for execution as soon as all its predecessors have finished execution and their transmitted data has been received, similar to a Kahn Process Network [Kah74]. Communication volume $v_{a,b}$ is set to zero if both nodes n_a and n_b are executed on the same core. Otherwise, the communication causes a delay in the NoC (see Section 3.1). The application is instrumented to allow the runtime system to monitor the performance of the application, similar to the work presented in [HES+10]. These measurements are used to adapt the parameters of the application performance model, as shown in Section 4.6.

4.3.2. Influence of the topological location of resources

Based on the communication delay equation given in Equation (3.2), this thesis proposed to improve the accuracy of application performance estimation based on the average number of hops between the cores in C: the sum of the number of hops $w(c_a, c_b)$ between any two cores c_a, c_b divided by the number of these combinations Equation (4.6a).

$$w_{avg}(C) = \frac{\sum_{c_a \in C_{A_i}} \sum_{c_b \in C_{A_i} \setminus \{c_a\}} w(c_a, c_b)}{|C| \cdot (|C| - 1)} \tag{4.6a}$$

$$w_{avg}(\mathcal{C}) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} w(c_i, c_j)}{N \cdot (N-1)} = \frac{2}{3}\sqrt{N} \tag{4.6b}$$

The average number of hops in C depends on the selection of the cores. For instance for $|C| = 40$ out of $N = 256$ cores, the difference between the best case (spatially as close to each other; 4.1 hops on average) and the worst case (spatially as far as possible from each other; 14.5 hops on average) is significant. The values of $w^{\#,min}_{avg}(k)$ for the best case selection of cores in C (see Equation (4.7a)), and $w^{\#,max}_{avg}(k)$ for the worst case selection (see Equation (4.7b)) are shown in Figure 4.4.

$$w^{\#,min}_{avg}(k) = min\left\{ w_{avg}(C) \,\middle|\, C \subseteq \mathcal{C}, |C| = k \right\} \tag{4.7a}$$

$$w^{\#,max}_{avg}(k) = max\left\{ w_{avg}(C) \,\middle|\, C \subseteq \mathcal{C}, |C| = k \right\} \tag{4.7b}$$

To obtain $w^{\#,min}_{avg}(k)$ and $w^{\#,max}_{avg}(k)$, C_{A_i} is iteratively extended by the core with the lowest/highest spatial distance to the cores already included in C, starting from

the core topologically in the middle of the system for the best-case respectively the core in the top-left corner in the worst-case. Mappings should achieve $w_{avg}(C)$ close to the respective best case $w_{avg}^{\#,min}(k)$ to optimize application performance. The worst-case average distance $w_{avg}^{\#,max}(k)$ represents the worst-case selection of n cores in C that might occur when mapping the application, e.g. in a system that is heavily loaded with other concurrent applications.

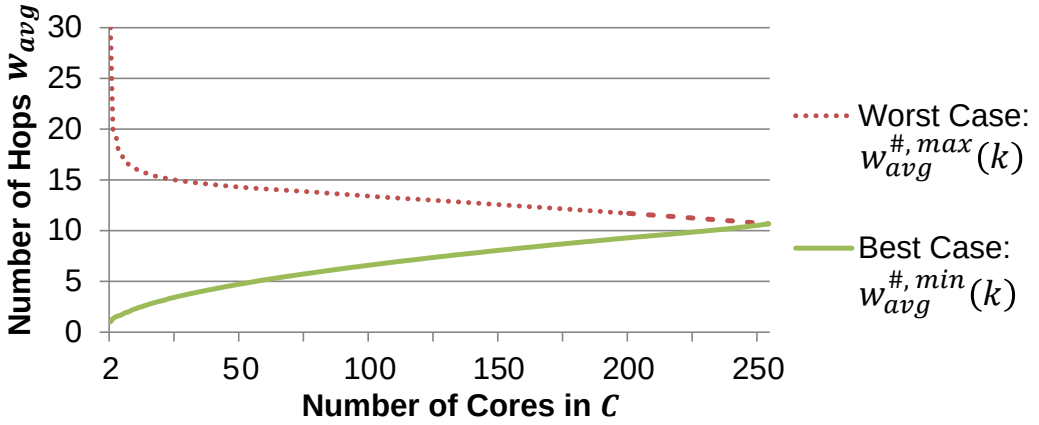

Figure 4.4.: Average number of hops $w_{avg}(C)$ required to send a message between two cores when allocating a subset C of all N=256 cores to application A_i a) in the best case, and b) in the worst case

Figure 4.5 shows the influence that different sets of cores C_{A_i} have on the achievable speedup for tree different applications. The first task-graph represents an application with sparse communication. Each node in the task-graph (except the end node per block) only depends on a few predecessors, which allows the task mapper to map the tasks in a way that avoids frequent communications across different cores. The second application features a medium communication density. There are up to five inputs to each node required before the node can execute. This also means that the topological location of the allocated cores has a bigger influence on the speedup. The third application is based on execution traces of a real world robotic vision application ([PSK+12], see Appendix B). Figure 4.5a), c), and e) show the influence of the average number of required hops $w_{avg}(C_{A_i})$ on the achievable speedup. The average number of required hops $w_{avg}(C_{A_i})$ between the cores in C_{A_i} almost linearly correlates with the achieved speedup. The speedup curves $S_{A_i}^{\#,best}(k)$ and $S_{A_i}^{\#,worst}(k)$ resulting from executing the applications on the best case (see Equation (4.8a)) respectively worst case (see Equation (4.8b)) selections of cores in C are shown in Figure 4.5b), d), and f).

$$S_{A_i}^{\#,best}(k) = max\left\{S_{A_i}(C) \,\middle|\, C \subseteq \mathcal{C}, |C| = k, w_{avg}(C) \widehat{=} w_{avg}^{\#,min}(k)\right\} \tag{4.8a}$$

$$S_{A_i}^{\#,worst}(k) = min\left\{S_{A_i}(C) \,\middle|\, C \subseteq \mathcal{C}, |C| = k, w_{avg}(C) \widehat{=} w_{avg}^{\#,max}(k)\right\} \tag{4.8b}$$

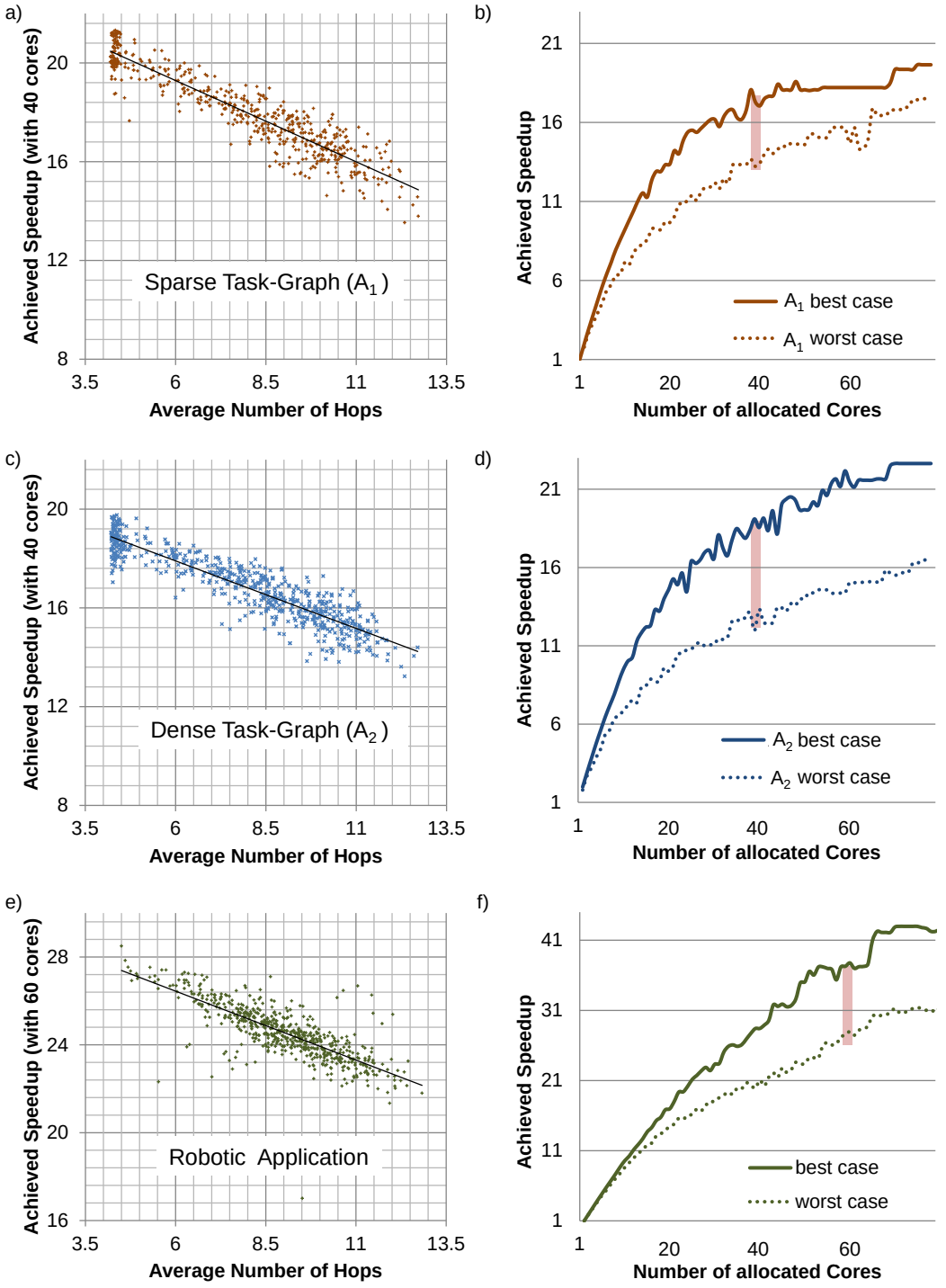

Figure 4.5.: Influence of different metrics [average number of hops in a), c), e) and number of allocated cores in b), d), e)] on the achievable speedup compared to execution on a single core for a many-core system of size N=256

4.4. Considering the Topology for Speedup Estimation

Based on the almost linear relationship between the achieved speedup of an application A_i and the average number of hops $w_{avg}(C)$ between the cores in C for any number of cores $k = |C|$, this thesis proposes to use this simple metric to allow the performance model to consider the relative topological location of the cores C to be allocated to A_i. As the influence of $w_{avg}(C)$ on the speedup is different for each application, the two corner-cases are combined: the highest achievable speedup $S_{A_i}^{\#,best}(k)$, and the worst case speedup $S_{A_i}^{\#,worst}(k)$. To estimate the actual performance of A_i, a linear interpolation between both values based on the value of $w_{avg}(C)$ is performed, as shown in Equation (4.9).

$$k = |C|$$
$$S_{A_i}(C) = S_{A_i}^{\#,worst}(k) + \frac{w_{avg}^{\#,max}(k) - w_{avg}^{\#,min}(k)}{w_{avg}(C) - w_{avg}^{\#,min}(k)} \cdot \left(S_{A_i}^{\#,best}(k) - S_{A_i}^{\#,worst}(k)\right) \tag{4.9}$$

To efficiently represent the speedup curves $S_{A_i}^{\#,best}(k)$, and $S_{A_i}^{\#,worst}(k)$ the application performance model introduced by Downey [Dow98] is used, as it captures the behavior of real applications running on real parallel machines very well, see Section 4.2.2 It uses two parameters (the average parallelism P of an application and its variance in parallelism σ) to describe typical application speedup curves, similar to the ones shown in Figure 4.5b), d), and f). Downey's application model does not consider the topological location of the cores, i.e. it is topology-agnostic.

4.5. Offline Parameterization of Model Parameters

To initialize the model parameters, applications are individually profiled offline. Therefore, application A_i is executed on different sets of cores C_{A_i} starting from one core ($|C_{A_i}| = 1$) through all cores in the system ($C_{A_i} = \mathcal{C}$). For each number of cores $k = |C_{A_i}|$ the cores are once selected to be spatially as close as possible to each other to obtain $S_{A_i}^{\#,best}(k)$ and once to be spatially as far as possible from each other to obtain $S_{A_i}^{\#,worst}(k)$. Then Downey's model is parameterized to match the observed $S_{A_i}^{\#,best}(k)$ and $S_{A_i}^{\#,worst}(k)$. The parameters for Downey's model are stored in the parameter tuple $(P_{A_i}^{best}, \sigma_{A_i}^{best}, P_{A_i}^{worst}, \sigma_{A_i}^{worst})$ for each block (see Figure 4.3) of the application. The offline analysis obviously does not cause runtime overhead. Applications may be profiled at any time before they are executed on the system, i.e. it is not required to know all applications at design time.

4.6. On-the-fly Adaptation of Model Parameters

To improve the accuracy of the performance predictions, the parameters ($P_{A_i}^{best}$, $\sigma_{A_i}^{best}$, $P_{A_i}^{worst}$, $\sigma_{A_i}^{worst}$) of the application performance model are adapted to react to spontaneous workload variations. Therefore, the measured performance of multiple executions is used to adapt the parameters of the model in a way that minimizes the error between the measured speedup $S_{A_i}^{measured}(C_{A_i})$ (see Section 4.6.1) and the estimated speedup $S_{A_i}(C_{A_i})$. Figure 4.6 gives a high-level overview on the on-line adaptation of the model parameters.

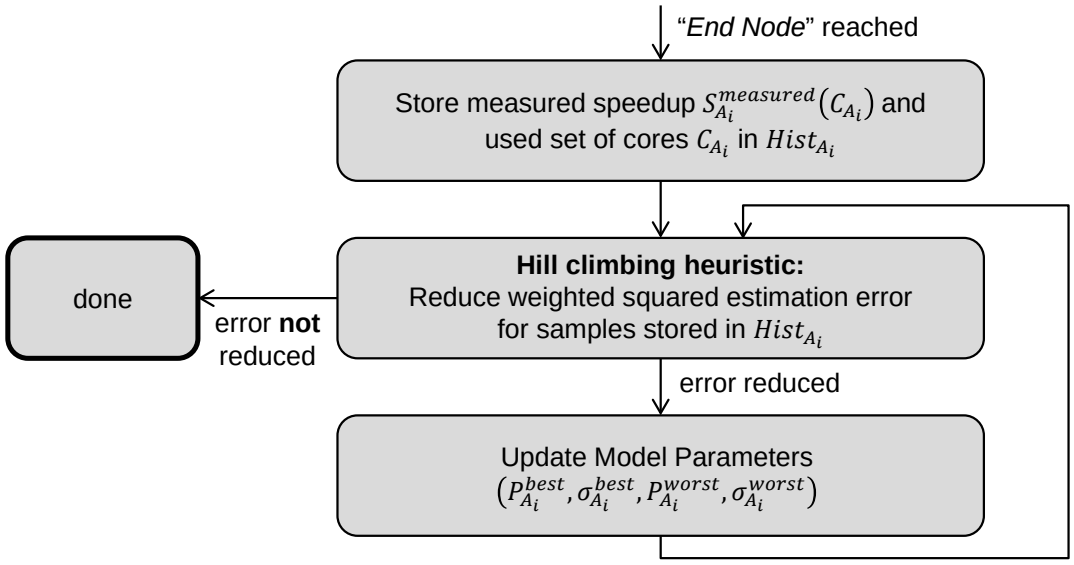

Figure 4.6.: High-level overview on the on-line adaptation of model parameters

4.6.1. Monitoring of Application Performance

The popular heartbeat framework [HES+10] was adapted to measure the time required by the application to complete each phase. Applications emit heartbeats in the *start* and *end* nodes that flank each phase. The time that passed between those two heartbeats is the execution time $T_{A_i}(C_{A_i})$. The execution time $T_{n_x} \approx w_x$ of each node n_x is determined by subtracting its start time from its end time, to obtain the momentary dynamic workload of the application. Whenever the end node triggered its heartbeat, $S_{A_i}^{measured}(C_{A_i})$ is calculated according to Equation (4.10).

$$S_{A_i}^{measured}(C_{A_i}) = \frac{\sum_{n_x \in A_i} T_{n_x}}{T_{A_i}(C_{A_i})} \tag{4.10}$$

$S_{A_i}^{measured}(C_{A_i})$ and C_{A_i} then are stored in the sample history Hist_{A_i}, and the model parameters of S_{A_i} are adapted (see Section 4.6.2) to improve the accuracy of subsequent estimations of $S_{A_i}(C')$ for different C'.

4.6.2. Heuristic Adaptation

Due to the monotonic increasing behavior of the speedup curves $S_{A_i}^{\#,best}(k)$ and $S_{A_i}^{\#,worst}(k)$ represented with Downey's performance model, a fast hill climbing heuristic can be used to adapt the parameters at runtime. In the following, the parameter tuple $(P_{A_i}^{best}, \sigma_{A_i}^{best}, P_{A_i}^{worst}, \sigma_{A_i}^{worst})$ is used interchangeable with S_{A_i}.

The goal of the adaptation (Listing 3) is to minimize the weighted estimation error that is determined by comparing the measured values $S_{A_i}^{measured}(C_{A_i})$ with $S_{A_i}(C_{A_i})$ for the samples stored in Hist_{A_i} (Listing 4). To focus on recent changes in the workload, the weight of older samples is reduced exponentially. To avoid miss-adaptations caused by an empty sample history Hist_{A_i}, measurements from previous executions of the application remain in the history.

The actual adaptation works by repeatedly selecting a new configuration S'_{A_i} based on S_{A_i} that reduces the estimation error until no further error reduction is achieved or the step width δ is below a threshold. One of the model parameters is either increased or decreased, resulting in eight possible ways to configure S'_{A_i} based on S_{A_i}. The configuration S'_{A_i} with the lowest estimation error for the values in Hist_{A_i} is chosen as the new configuration for S_{A_i} if it resulted in less error than the current configuration. The step width δ by which the parameters are changed is reduced in each iteration to allow for a fast adaptation of the parameters first and a fine-tuning of the parameters in the end. As the meaningful range of the parameters σ and P differs, the value δ by which they are changed is fit by the constant factors W_σ and W_P.

4.7. Performance Model Evaluation

In this section, the on-the-fly estimation accuracy of the application performance model (APM) as well as the induced runtime overhead is evaluated to show its ability to adapt to spontaneous workload variations which results in highly efficient utilization of many-cores. The centralized hill-climbing heuristic presented in Section 5.3.1 is used to evaluate the model. The achieved results are compared with state-of-the-art multi-application mapping for many-core systems [YGSP13] (presented in detail in Section 5.1.2) as well as Downey's topology agnostic application performance model [Dow98]. The introduced overhead is discussed. Additional results are presented in Chapter 6.

Listing 3: Runtime Model Adaptation

Input: S_{A_i} as ($P^{best}_{A_i}$, $\sigma^{best}_{A_i}$, $P^{worst}_{A_i}$, $\sigma^{worst}_{A_i}$)
Output: Updated S_{A_i}

$W_\sigma \leftarrow 0.1$; `// step width multiplier for parameters` σ
$W_P \leftarrow 1$; `// step width multiplier for parameters` P
$\delta \leftarrow 1$; `// initial step width`
gain $\leftarrow 1$;
`// Now use a hillclimbing heuristic to match the parameters to the observed values`
while *gain* > 0 **and** $\delta > 0.2$ **do**
  `// assemble the eight neighboring configurations` NC
  $NC \leftarrow (P^{best}_{A_i} \pm \delta \cdot W_P, \sigma^{best}_{A_i}, P^{worst}_{A_i}, \sigma^{worst}_{A_i})$;
  $NC \leftarrow NC \cup (P^{best}_{A_i}, \sigma^{best}_{A_i} \pm \delta \cdot W_\sigma, P^{worst}_{A_i}, \sigma^{worst}_{A_i})$;
  $NC \leftarrow NC \cup (P^{best}_{A_i}, \sigma^{best}_{A_i}, P^{worst}_{A_i} \pm \delta \cdot W_P, \sigma^{worst}_{A_i})$;
  $NC \leftarrow NC \cup (P^{best}_{A_i}, \sigma^{best}_{A_i}, P^{worst}_{A_i}, \sigma^{worst}_{A_i} \pm \delta \cdot W_\sigma)$;
  `// select the neighboring configuration that results in the lowest re-estimation error`
  $S'_{A_i} \leftarrow \text{argmin}_{S \in NC} \left\{\text{HistoryError}_{A_i}(S)\right\}$; `// see Listing 4`
  gain $\leftarrow \text{HistoryError}_{A_i}(S_{A_i}) - \text{HistoryError}_{A_i}(S'_{A_i})$;
  if *gain* > 0 **then**
    $S_{A_i} \leftarrow S'_{A_i}$
  end
  $\delta \leftarrow 0.9 \cdot \delta$; `// exponentially decrease step width`
end
return S_{A_i};

Listing 4: History Sample Model Re-Prediction Error

Input: Speedup estimation function S_{A_i}, chronologically ordered sample history Hist_{A_i}
Output: Exponentially weighted sum of the squared estimation errors
error $\leftarrow 0$;
$\alpha \leftarrow 1$; `// initial weight`
foreach *Sample* $s \in Hist_{A_i}$ **do**
  error $\leftarrow$ error $+ \alpha \cdot \left(S_{A_i}(s.C_{A_i}) - s.S^{measured}_{A_i}(s.C_{A_i})\right)^2$;
  $\alpha \leftarrow 0.9 \cdot \alpha$; `// exponentially decrease the weight`
end
return error;

All results were obtained by mixed-level simulation of the NoC based many-core system with $N = 256$ (16x16) cores. Computation is simulated at the abstraction level of task-graphs, i.e. at a much more detailed level than the on-the-fly application performance estimation model. Communication is simulated for a NoC that employs XY-Routing and considers link congestion for an accurate and mapping-dependent simulation (see Appendix A). The performance-effective and low-complexity task scheduling scheme presented in [THW02] was used to a) map the tasks of the application to the cores in C_{A_i} and to b) schedule their execution. The scheduler selects the task $n_x \in A_i$ with the highest upward rank value (the length of the critical path to the exit task, including computation costs and communication costs) and assigns that task to the core that minimizes the earliest finish time of the task. The selection of the core the start task is mapped to had to be extended, as this selection has noticeable impact on the applications' performance. For the sake of simplicity, the schedule is calculated with each core in C_{A_i} set as the core c_{start} the start task is mapped to once. Eventually, the selection of c_{start} which resulted in the shortest schedule with the earliest finish time was chosen.

4.7.1. Estimation Accuracy

To evaluate the accuracy of the presented application performance model, three different application scenarios plus a full robotic vision application on various allocations of cores C. The scenarios were generated with TGFF [DRW98]. The total workload of the three applications and the number of tasks in each application is almost the same; they only differ in their communication density. The first task-graph represents an application with sparse communication. Each node in the task-graph (except the end node per phase) only depends on a few predecessors, which allows the task mapper to map the tasks in a way that avoids frequent communications across different cores. The second application features a medium communication density. There are up to five inputs to each node required before the node can execute. This also means that the topological location of the allocated cores has a bigger influence on the speedup. The third task-graph represents an application with very dense communication between the individual tasks. For this task-graph, the relative topological location of the allocated cores is of upmost importance. As a real-world application example, traces of a robotic vision application [PSK+12] were used as a malleable software pipeline [JPK+13] (see Section 2.2.4 and Appendix B for more details).

The application performance model is compared with Downey's topology-agnostic application performance model from [Dow98]. Additionally, very accurate (but computationally intensive) estimates of the achievable speedup are obtained by mapping the application tasks to the allocated cores [THW02] and using the offline profiled execution times to determine the expected execution time of the applications. To compare the accuracy of these three estimates, the estimates are compared to the

real measured speedup $S_{A_i}^{measured}(C_{A_i})$ (obtained as described in Section 4.6.1).

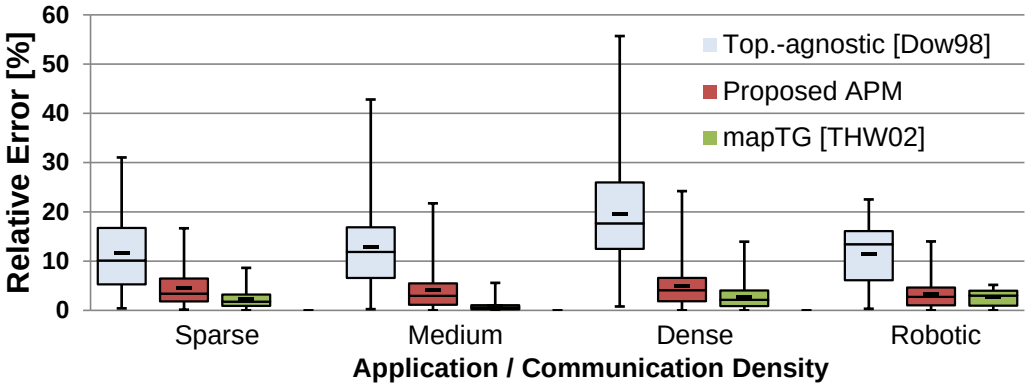

Figure 4.7.: Relative estimation error made when estimating the speedup of the same application A_i executed on random allocations of cores C_{A_i} when using the topology-agnostic model [Dow98], the proposed adaptive on-the-fly estimation model, and the computational intensive estimation by mapping the applications task-graph [THW02].

Figure 4.7 shows the relative error of estimations (note that the mapping efficiency will be compared with MAMS [YGSP13] later on, but as MAMS does not perform speedup estimates, it cannot be compared here). The instances of C were selected randomly. In all cases, the accuracy of the proposed model is significantly better compared to using the topology-agnostic estimate [Dow98] that does not consider the relative topological location of the allocated cores. At comparable computational effort (evaluated in Section 4.7.2), the proposed application performance model predicts the speedup with an average relative error of 4.5% compared to 14.7% [Dow98]. The computational intensive (see Section 4.7.2) estimation [THW02], achieved by mapping the applications task-graph to the different allocations C_{A_i}, still results in an average relative error of 1.9% caused by dynamic execution behavior. The maximum estimation error is significantly lower than the errors that occur with the topology-agnostic speedup estimation, leading to more efficient application mappings, as shown in Section 4.7.4.

4.7.2. Overhead Analysis

An application performance model that is suitable for runtime resource management should have low computational effort to allow for evaluating many different resource allocations on-the-fly (e.g. thousands of different resource allocations in tens of milliseconds) for frequent adaptations of the application mapping. Determining the best-case speedup $S_{A_i}^{\#,best}(|C|)$ and the worst-case speedup $S_{A_i}^{\#,worst}(|C|)$ requires only a small number (≤ 10) of additions/multiplications and thus completes in a few CPU cycles. An ample computation of $w_{avg}(C)$ is within $\mathcal{O}(n^2)$ with $n = |C_{A_i}|$.

In practice, often only one core is added to or removed from C such that the complexity of determining $w_{avg}(C)$ is reduced to $\mathcal{O}(n)$ by adding/removing only the connections to the added/removed core. This results in an execution time that is approximatively n times higher than the topology-agnostic model [Dow98], but compared to the execution time of the task-mapping heuristic [THW02] it is negligible. The execution time of the implementations was measured on an Intel i5-2500 CPU. On average, for estimating the speedup of an application running on 40 cores, Downey's topology-agnostic model [Dow98] required 31ns (i.e. about 100 cycles) and the proposed application performance model required 1208ns. The task-mapping heuristic [THW02] required even 2ms, a factor 2000 slower per speedup estimation. The execution time does not depend on the number of cores in the system; only the number of cores that is assigned to an application has an influence – additional measurement results for 10 cores and 100 cores are shown in Figure 4.8.

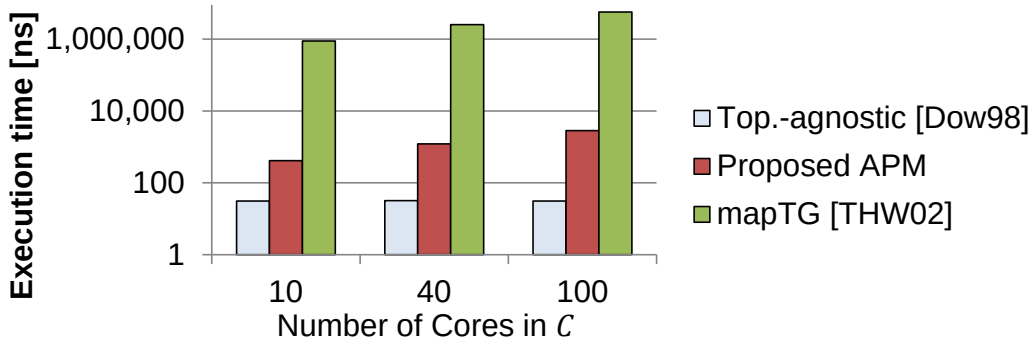

Figure 4.8.: Time required to estimate the application speedup by the topology-agnostic model [Dow98], the proposed adaptive on-the-fly application performance model and the exact solution that actually maps the applications tasks to cores [THW02]

Whenever an application completed execution of a phase (see Section 3.2), the model parameters are adapted. The adaptation needs to perform at most 15 iterations of the outer loop in which eight configurations of $S_{A_i}(C)$ are evaluated against the measured samples stored in $Hist_{A_i}$. As $w_{avg}(C_{A_i})$ is already stored in $Hist_{A_i}$, the computational complexity of $S_{A_i}(C)$ is reduced to the complexity of Downey's application performance model. Therefore, the adaptation overhead is within $\mathcal{O}(|Hist_{A_i}| \cdot 15 \cdot 8)$. By limiting $|Hist_{A_i}|$ to 10, adaptation can be finished within 50µs, which is practicable and feasible.

4.7.3. Evaluating Application Mappings

As shown in Section 4.2, there is no directly comparable model to estimate the speedup of an application given a potential allocation of cores C that also considers

the topological properties of these cores. However, there are runtime application-mapping approaches for many-core systems that try to optimize the communication latencies without explicitly estimating the application performance. The Multi-Application Multi-Step (MAMS) mapping method for many-core systems presented in [YGSP13] is used as reference implementation as it has a similar optimization goal and achieves state-of-the-art application mapping quality. See Section 5.1 for additional details. The implicit optimization of the communication latencies is achieved by focusing on rectangles as square as possible. MAMS only optimizes for the topological location – to make the results comparable, MAMS was improved to determine the ideal number of cores for each application by using the static topology-agnostic application performance model presented in [Dow98]. In most cases the allocation of rectangles for each application leads to a fragmentation of the application mapping, i.e. there are cores left over that cannot be allocated to an application. As the application mapping resulting in using the other application models consumes all available cores, the MAMS mapping method was further improved in the way that, after the rectangles for each application have been defined, the unallocated cores were distributed to spatially adjacent applications. Again, the performance model presented in [Dow98] is used to decide on which of the adjacent applications would benefit the most from additional cores.

4.7.4. Adaptation to Workload Variations

The presented mapping heuristics are used to evaluate the benefits of the improved accuracy and the adaptability of the proposed speedup estimation model for various combinations of different applications in different scenarios. These scenarios have been selected to demonstrate the adaptability of the application performance model as well as the broad applicability to different and typical kinds of system utilization. The speedup is determined through simulating the applications. As comparison metric, the efficiency of the application mapping is obtained through the average speedup per core (see Equation (4.11)).

$$\text{Mapping Efficiency} E = \frac{\sum_{i=1}^{M} S_{A_i}\left(C_{A_i}\right)}{N} \tag{4.11}$$

In the first two scenarios, the number of concurrent applications is gradually increased from 15 to 28 (Figure 4.9) respectively reduced from 30 to 17 (Figure 4.10). The resulting application mappings obviously benefit from the more accurate estimates of the proposed adaptive model. Due to the hill climbing behavior of the mapping heuristic (see Section 5.3.1) and the not monotonically increasing shape of the speedup curve determined by using the task-mapping heuristic [THW02] (see Figure 4.5b), d), and f)), the resource-aware adaptive estimates sometimes even lead to slightly more efficient mappings. The application mappings for these two scenarios

are 8.0% more efficient when using the proposed application performance estimation model compared to state-of-the-art (MAMS) runtime mapping [YGSP13].

In the third and fourth scenario, the number of applications decreases abruptly (from 30 to 15 concurrent applications, Figure 4.11) respectively increases abruptly (from 15 to 30 concurrent applications, Figure 4.12). Again, the resulting application mappings achieved when using the proposed adaptive speedup model are almost always better than the mappings from MAMS. The adaptation to the new operating conditions is clearly visible, i.e. the application mapping efficiency increases with each iteration after the abrupt workload change. In these two scenarios, on average the application mapping achieved by using the adaptive estimation model resulted in a 4.3% improvement compared to MAMS.

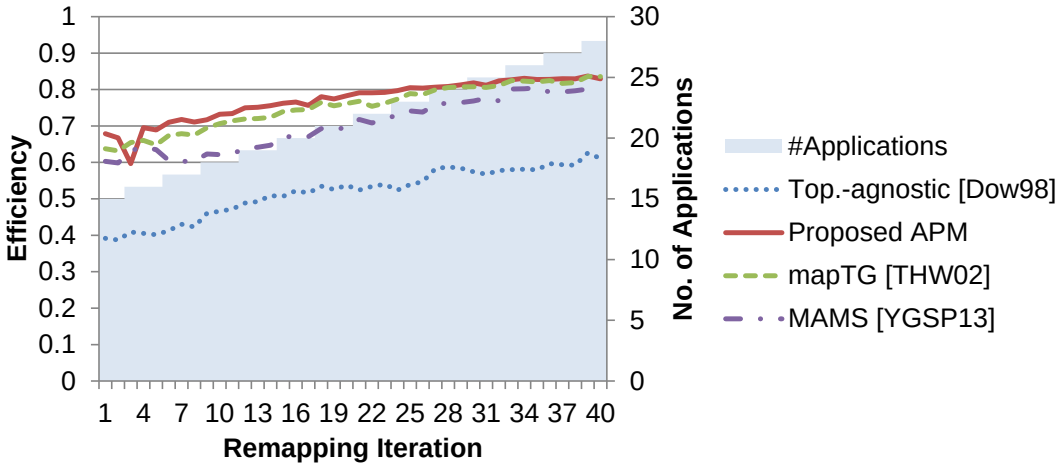

Figure 4.9.: Application mapping efficiency for an increasing number of concurrent applications: A new application is started every third iteration – starting from 15 to 28 concurrent applications in the end

In the fifth scenario, the workload includes periodic bursts, just as they are common in embedded control systems. Figure 4.13 shows the efficiency of the system utilization over the time of four such bursts. Compared to MAMS, there is a slight advantage in the time between the bursts when using the resource-aware speedup model to decide on the application mapping. However, the workload within the bursts is handled 6.8% more efficient, on average. Again, in all cases the topology-agnostic speedup estimation results in the worst mappings.

For the sixth scenario (shown in Figure 4.14), the mapping efficiency is analyzed for random changes in the workload. Such a workload is common for interactive system utilization. Especially in the cases of multiple concurrent applications, mappings achieved through the adaptive on-the-fly performance model outperform state-of-the-art [YGSP13]. This is because not all applications necessarily benefit to the same degree from spatially close cores. In contrast to MAMS, the proposed model allows some applications to use slightly scattered sets of cores to allow allocating larger

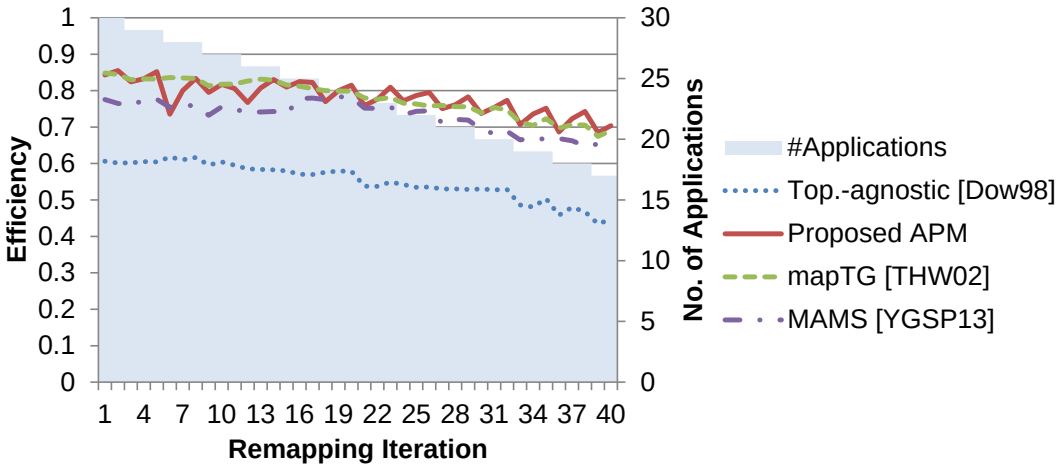

Figure 4.10.: Application mapping efficiency for a decreasing number of applications: An application is terminated every third iteration – starting from 30 to 17 concurrent applications in the end. The adaptation of the proposed model is clearly visible in the form of small improvements after each change in the workload

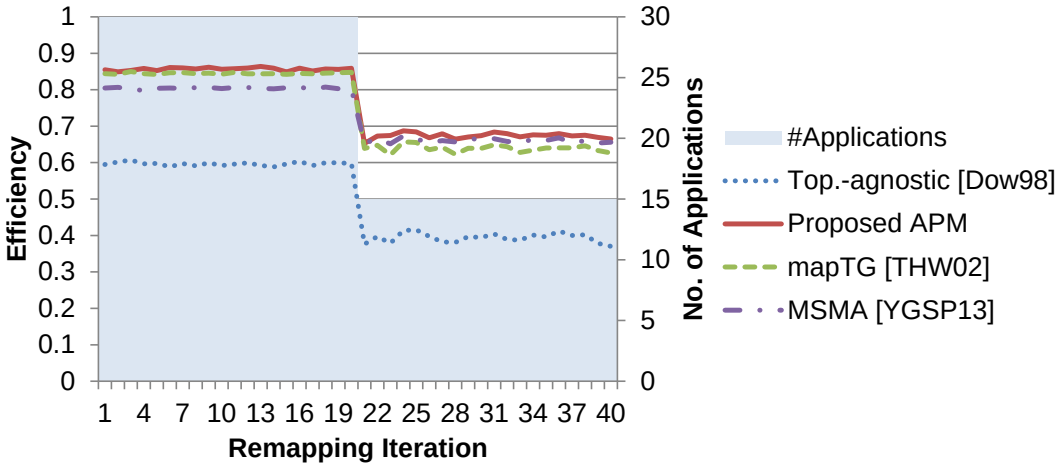

Figure 4.11.: Application mapping efficiency for an abrupt change (the number of applications is reduced from 30 to 15) in the workload. After the workload change the parameters of the proposed model are on-the-fly adapted to the new workload situations

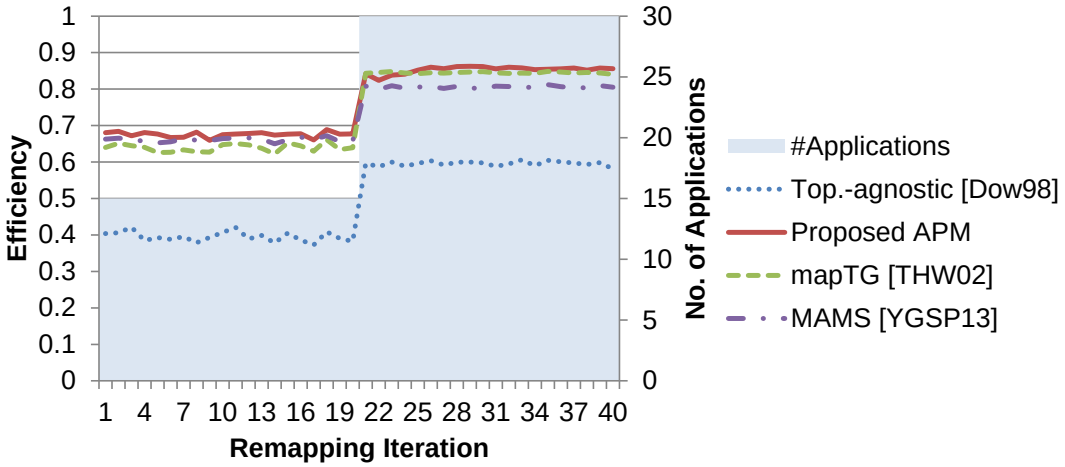

Figure 4.12.: Application mapping efficiency for an abrupt change (the number of applications is increased from 15 to 30) in the workload. Again, the adaptation of the model parameters is clearly visible

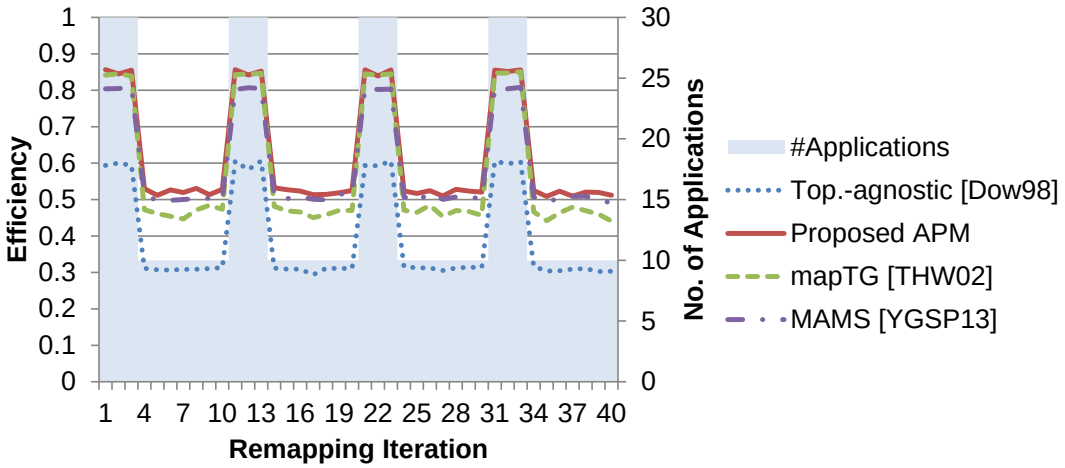

Figure 4.13.: Application mapping efficiency for periodic bursts in the workload – e.g. as experienced in embedded control systems

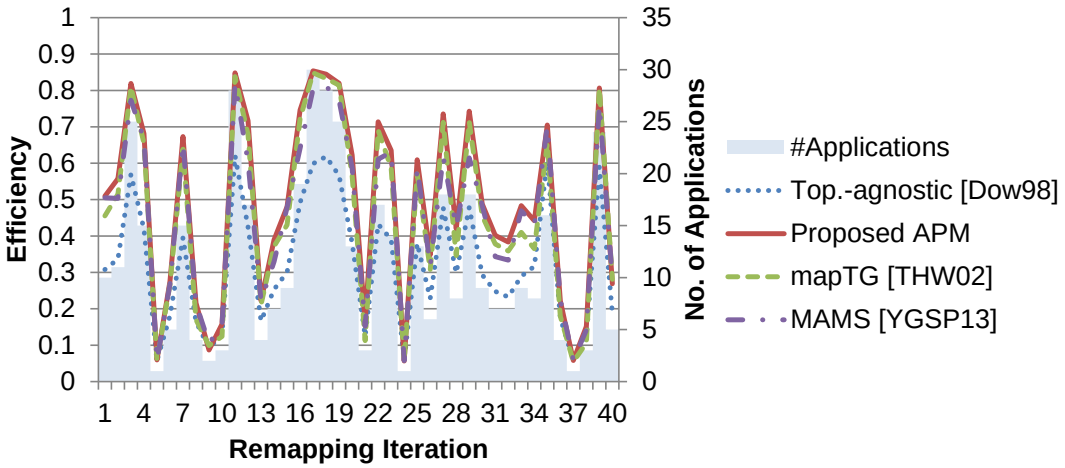

Figure 4.14.: Application mapping efficiency for random changes in the workload – e.g. as experienced in interactive system utilization

coherent sets of cores to those applications that benefit more from the smaller average number of hops. On average, the adaptive on-the-fly estimation results in 7.4% more efficient application mappings than when using MAMS and it provides almost the same mapping quality compared to the most accurate but computationally unfeasible estimation that calculates the task-graph mapping for each estimation.

On average for the presented scenarios, the application mapping efficiency is improved by 6.4% compared to state-of-the-art runtime application mapping for many-cores [YGSP13]. The resulting application mappings are significantly (32%) more efficient than the application mappings that resulted in using the topology-agnostic speedup estimation model [Dow98].

4.8. Summary of Application Performance Modeling

An adaptive on-the-fly application performance model has been presented. The presented results have demonstrated that the accuracy of the performance estimations results in overall high execution efficiency when using application performance estimates for application mapping decisions. The evaluations show that the average estimation error is reduced from 14.7% to merely 4.5% while at the same time significantly reducing the worst-case error. As a result, the applications profit from better mappings compared to state-of-the-art. The work enables managing many-core systems that exhibit rapid and spontaneous workload variations while still maintaining high mapping quality. The on-the-fly application performance estimation is used by the distributed resource management presented in the next Chapter and due to its low overhead allows to keep resource management latency low.

5. Runtime Resource Management for Many-Cores

This Chapter is based on the following publication:

[KBL+11] S. Kobbe, L. Bauer, D. Lohman, W. Schröder-Preikschat, and J. Henkel. DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, 2011.

Mapping multiple applications to a set of cores and adapting each application to the allocated cores (e.g. by choosing a different algorithmic implementation or by increasing/decreasing the degree of parallelism) is key for an efficient utilization of the computation resources in many-core systems. Mapping applications means to (re-)allocate a subset of cores $C_{A_i} \subseteq \mathcal{C}$ to each application A_i such that no core is assigned to two applications at the same time, i.e. for all A_i, A_j with $i \neq j$, $C_{A_i} \cap C_{A_j} = \emptyset$. The goal of runtime resource management is to dynamically select these subsets C_{A_i} in a way that follows a predefined optimization goal (see Section 1.2).

Systems with hundreds or thousands of cores integrated on a single chip (see Section 2.1) span a huge solution space – the problem of optimal mapping of parallel applications to cores is known to be NP-complete [CGJ78]. The size of the search space for an optimal mapping grows factorial with the number of cores [MMCM07]. With high dynamic workloads (i.e. resource demands that are not known a priori), these mappings cannot be predetermined and need to be decided online.

The challenge in resource management is to a) achieve high application mapping quality with respect to the optimization goal while at the same time b) keeping the associated overhead and the latencies of computing the application mapping low. In general, runtime resource management should find an efficient trade-off between application mapping quality and the resource management overhead introduced to the system.

This Chapter presents the distributed resource management developed in the scope of this thesis which is able to flexibly react to changes of the resource demands of applications by adaptively selecting the employed resource management strategy. Thereby, it is able to find a good initial mapping of applications fast and to optimize

the application mapping gradually with a very low overhead. It uses the concept of a Multi-Agent-System (see Section 2.3) to achieve scalability by focusing on local decision making based locally available information. The Chapter first gives an overview on related work (Section 5.1) and on the targeted optimization goals (Section 5.2). It then shows how mapping decisions may be made in a centralized fashion (Section 5.3). The resulting latency from centralized decision making is analyzed to motivate the need for scalable resource management. In the rest of this Chapter, the distributed resource management for many-cores that has been developed in the scope of this thesis is presented.

5.1. Related Work

A large body of research work has been conducted in the area of resource management for many-core systems. These works can be classified into offline (e.g. [JSCT08, MMCM07, KKJC02, BK90]), mixed approaches that utilize offline pre-calculated mappings that are selected at runtime (e.g. [SBR+12, HMS+11, SGB+09]), and online approaches (e.g. [ATBS13, YGSP13, SLS07, DA06]).

If the workload of a system is known at design time and if it does not change at runtime, the mapping can be decided offline for malleable as well as for fixed-size applications (see Section 2.2). An optimal (e.g. [JSCT08]) or near-to-optimal solution can be found by applying exhaustive and stochastic search methods and heuristic approaches [MMCM07, KKJC02]. To speed up the offline search, parallel approaches have been presented [BK90].

However, all offline approaches are based on a priori knowledge about all possible system states and thus they cannot react to unforeseen situations or interactive operation of the system. Especially if the workload varies dynamically [CCD+08], offline approaches cannot be applied.

Mixed approaches address this situation by pre-calculating possible application mappings and then select the appropriate mapping at runtime. This also means that all applications that should be executed and all possible combinations thereof must be known a priori, however, the sequence of execution can vary. In [SGB+09], a compositional heuristic to solve the multi-dimensional multiple-choice knapsack (MMKP) problem, that models application mapping, is presented. The heuristic is based on an offline analysis of the Pareto optimal mappings for all applications. There is one central instance in the system that selects the best application mapping from the pre-calculated solutions at runtime. Dynamic workloads (i.e. applications that dynamically change their resource demands [CCD+08]) are not supported. In [SBR+12], the "distributed application layer" is presented which is scalable as it uses a hierarchy of monitoring and controlling infrastructure. However, all application mappings are offline pre-calculated for any combination of applications that should be executed. Therefore, the approach is also unsuitable for dynamic workloads.

Pure online application mapping approaches are able to react to changes in the workload of the system by adjusting the application mapping for combinations of different applications that may not be known at design-time, i.e. the application mapping is not predetermined but decided at runtime. Usually, online mapping approaches trade off application mapping quality against scalability and computational feasibility. If the number of cores in the many-core system allows for centralized resource management, approaches like [SLS07, YGSP13] achieve considerably good application mappings. However, their computational complexity hinders their utilization for many-core systems. Parallel approaches utilize multiple cores to speed up the application mapping. For instance [SS11] relies on a parallel binary search to decide the number of cores to allocate to each application, however, it only decides on the number of cores to allocate to each application and does not consider the topology of the resources. In the following Subsections, selected approaches for online resource management from different domains are presented.

5.1.1. Grid- and Cloud-Computing

The idea behind grid computing is to make computation resources of many different and often geographically distributed computers available to highly parallel applications and to create a big virtual computation environment [BFH03, FK03]. The applications typically stem from the domain of scientific computing. To use a grid for an application, the computation must be separable into many individual work packages that then are scheduled to the computers allocated to the application in the grid. This easily allows the applications to be malleable, which – despite the totally different computation platforms – creates similarities to the application model and also the resource management presented in this thesis.

Resource management in a grid is typically part of the middle-ware, whereof centralized as well as distributed approaches exist [DA06]. For instance AppLeS [BWC+03] (Application Level Scheduling) uses a distributed approach to allocated resources in a grid to applications. Each application aims at optimizing its own execution efficiency. Therefore AppLeS incorporates static and dynamic resource information, performance predictions, application and user-specific information, and scheduling techniques that adapt to application execution on-the-fly [BWC+03]. A brief overview on the steps involved in AppLeS is shown in Figure 5.1. An economy based bargaining for resources in a grid has been presented in Nimrod/G [BAG00] which tries to find sufficient resources to meet the application's deadline, and adapts set of computers it is using for each application depending on the competition among other applications for them.

Both (and most other resource managers for grid computing) rely on the Globus toolkit [Fos01], a layered architecture in which high-level global services are built on top of distributed local services, e.g. the Globus Resource Allocation Manager (GRAM) [CFK+98] that uses an abstract resource description language to perform

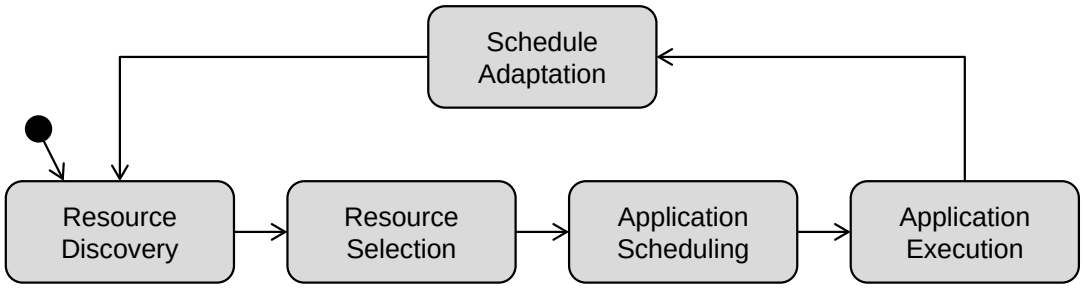

Figure 5.1.: Resource management performed per application in the AppLeS grid management. Resource discovery and selection rely on the Globus toolkit [Fos01] (adapted from [BWC+03])

resource allocation, or the Globus Metacomputing Directory Service (MDS) [FFK+97] that provides dynamic information on the state and structure of the grid to resource brokers like AppLeS or Nimrod/G for resource discovery. However, these approaches tend to involve a large computational effort and they demand many communication resources for resource management. This disadvantage of large overhead can be tolerated as in difference to the resource management developed in the scope of this thesis, application (re-)mappings are executed infrequently and applications tend to execute for multiple hours or even days. That is why these approaches are not applicable in highly dynamic systems where frequent changes in the workload (as expected for next-generation processing platforms [CCD+08]) are targeted.

The idea behind cloud computing [FZRL08] is different from grid computing – cloud computing provides on demand centrally managed computation resources to applications for a dynamic fee. Computation resources are made available in the form of virtual computers that are instantiated if the application requires additional resources and released afterwards. The mapping of virtual computers to physical resources is transparent to the application. Initializing a new virtual computer is not instantaneous; cloud hosting platforms introduce several minutes delay in the hardware resource allocation [IKLL12]. The virtual computers can be used to form a grid computing environment or to (temporary) extend an existing grid [OPF09]. Assuming that the cloud provider provides "unlimited" computation resources, the goal of resource management is no longer to provide the best performance to an application but to meet the desired application performance at the lowest cost. Resource management approaches for cloud computing (e.g. [ZA10]) typically optimize towards this goal by predicting the workload and adapting the resource allocation respectively.

5.1.2. The Multi-Application Multi-Step Mapping Method

The Multi-Application Multi-Step (MAMS) mapping method for many-core systems presented in [YGSP13] represents a centralized runtime resource management heuristic for many-core systems. Similar to the resource management presented in this thesis, MAMS first selects and allocates a set of cores C_{A_i} to each application A_i before the application tasks are mapped to these cores. MAMS itself does not support malleable applications – in Section 5.3.2, MAMS is combined with [SLS07] to realize a centralized resource management that supports malleable applications.

In [YXSP10] the weighted communication of an application (WCA) is defined as the sum of the products of the communication volumes of the messages sent between all tasks of the application A_i and the communication distance between the cores to which the tasks have been mapped. In [YXSP10], the WCA is used as optimization function to achieve a good application performance. However, determining the WCA of an application A_i given an allocation of cores C_{A_i} requires to first map the tasks to the cores, a computationally intensive operation that hampers the evaluation of many different allocations. MAMS [YGSP13] picked up the idea of minimizing the WCA to optimize the application performance, without mapping the application tasks. To achieve minimized WCAs, the selection of C_{A_i} works by finding empty rectangles on the chip that can accommodate the applications A_i. Cores within a rectangular area of the chip are spatially close to each other and therefore result in small WCAs. The implicit optimization of the communication latencies is achieved by focusing on rectangles as square as possible. With other words: the problem of application mapping is turned into the problem of managing empty rectangles on a 2d-mesh network, similar to the problem presented in [BKS00].

5.1.3. Distributed Management for Malleable Applications

The distributed resource management presented in [ATBS13] shares the application model, and the system model, and parts of the management protocol with the first incarnation of the distributed resource management DistRM [KBL+11] presented in this thesis. The application mapping is performed online, with support for malleable applications. The work presented in [ATBS13] does not rely on a central instance that facilitates application mapping. Instead, it uses multiple dedicated cores for resource management.

The many-core system is divided into multiple *regions* of same size. One dedicated core per region is responsible for managing the cores within the region that are momentary not allocated to an application and for maintaining a list of applications that do have cores allocated in the region. Additionally, each application uses one dedicated core to manage its own cores and the adaptation of the application to the allocated cores, similar to the Agents that are used in this thesis. The use of dedicated cores for resource management allowed to simplify the resource management protocol

and thus the communication overhead compared to DistRM [KBL+11]. However, Chapter 6 will show that due to the relative large size of the regions, the overall computation overhead is higher. Due to the dedication of multiple cores to resource management, less cores are available for application execution. [ATBS13] is evaluated in detail in Section 6.4.2.

5.2. Resource Management Optimization Goals

Resource management decisions following an global optimization goal. In this thesis, the decisions are steered by a so-called *utility function* which is used to calculate a comparable value for the utility $U_{A_i,P_j}(C_x)$ of a set of cores C_x for application A_i in phase P_j. The term utility function is adopted from economics and describes the "ability of a commodity to satisfy needs or wants". The notion of $U_{A_i}(C_x)$ refers to the utility of the core for application A_i in its momentary active phase. To compare different application mappings, the *mapping quality* $Q(A_1, A_2, \ldots, A_M)$ is used to express the overall application mapping quality, where a higher value of $Q(A_1, A_2, \ldots, A_M)$ means a better fulfillment of the targeted global optimization goal.

The optimization goal can be changed without modifying the presented protocols and algorithms, e.g. towards an optimization of energy consumption or, by including dynamic system monitoring information, towards an optimization of system health (e.g. by avoiding hot regions of the chip). Depending on the selected optimization goal, $U_{A_i}(C_x)$ and $Q(A_1, A_2, \ldots, A_M)$ have to be defined.

In the scope of this thesis, two goals are presented: a) the maximization of the average speedup of all concurrently executed applications (Section 5.2.1), and b) the minimization of the make-span of all concurrently executed applications (Section 5.2.2).

5.2.1. Maximization of the Average Speedup

The maximization of the average speedup of the concurrently executing applications leads to an efficient (efficiency is defined as the achieved speedup per core, see Equation (3.4)) operation of a many-core system, especially in cases with highly dynamic workloads. The average overall speedup $S^{average}_{A_1,A_2,\ldots,A_M}$ is calculated as shown in Equation (5.1), the resulting optimization goal is straight forward: maximize the average speedup as shown in Equation (5.2). The resulting mapping quality $Q^{S^{average}}(A_1, A_2, \ldots, A_M)$ (Equation (5.4) equals the average speedup $S^{average}_{A_1,A_2,\ldots,A_M}$ of all concurrently executing applications. The maximization of the average speedup of the concurrently executed applications leads to the reduction of the average turnaround time of applications. The turnaround time is one of the metrics used to

evaluate operating system scheduling algorithms [FR98] and is defined as the time span between the time $T_{A_i}^{ready}$ an application A_i is ready for execution and the time $T_{A_i}^{make}$ the application finished computation.

$$S_{A_1,A_2,\ldots,A_M}^{average} = \frac{\sum_{i=1}^{M} S_{A_i}\left(C_{A_i}\right)}{M} \tag{5.1}$$

$$\textbf{Optizimation Goal: } \text{maximize } S_{A_1,A_2,\ldots,A_M}^{average} \tag{5.2}$$

$$\textbf{Application Mapping Quality: } Q_{A_i}^{S^{average}}\left(C_{A_i}\right) = S_{A_i}\left(C_{A_i}\right) \tag{5.3}$$

$$\textbf{Overall Mapping Quality: } Q^{S^{average}}\left(A_1, A_2, \ldots, A_M\right) = S_{A_1,A_2,\ldots,A_M}^{average} \tag{5.4}$$

$$U_{A_i}^{S^{average}}\left(C_x\right) = \begin{cases} S_{A_i}\left(C_{A_i}\right) - S_{A_i}\left(C_{A_i} \setminus C_x\right) & \text{if } C_x \subseteq C_{A_i} \\ S_{A_i}\left(C_{A_i} \cup C_x\right) - S_{A_i}\left(C_{A_i}\right) & \text{if } C_x \nsubseteq C_{A_i} \end{cases} \tag{5.5}$$

The utility function $U_{A_i}^{S^{average}}\left(C_x\right)$ expresses the gain in speedup application A_i receives from using the cores C_x. It is directly based on the application performance model presented in Chapter 4. As shown in Equation (5.5) the utility function $U_{A_i}^{S^{average}}\left(C_x\right)$ is calculated by subtracting the achievable speedup of application A_i without using the cores in C_x from the achievable speedup when additionally using the cores in C_x. Depending on whether the set of cores C_x is already allocated to application A_i (i.e. $C_x \subseteq C_{A_i}$), C_x is imaginary removed from/added to C_{A_i}.

5.2.2. Minimization of the Make-Span

The make-span $T_{A_1,A_2,\ldots,A_M}^{make}$ of a set of applications $\{A_1, A_2, \ldots, A_M\}$ is defined as the latest completion of any of these applications, as shown in Equation (5.6). Minimizing the make-span (Equation (5.7)) is meaningful in batch-processing situations where a batch of applications is started at the same time and the completion of all applications is required in order to obtain the result or start the execution of the next set of applications. Obviously, the mapping quality $Q^{T^{make}}\left(A_1, A_2, \ldots, A_M\right)$ is better, the lower the make-span $T_{A_1,A_2,\ldots,A_M}^{make}$ is. As the mapping quality is defined as "higher is better", the reciprocal value of $T_{A_1,A_2,\ldots,A_M}^{make}$ is used to describe the mapping quality $Q^{T^{make}}\left(A_1, A_2, \ldots, A_M\right)$ as shown in Equation (5.9).

An optimization of the make-span is only possible, if all applications are able to estimate $T_{A_i}^{finish}(C_{A_i})$ based on the relative progress p_{A_i}, the total execution time $T_{A_i}^{\#}(1)$ of the application on one core, and the estimate of the momentary speedup $S_{A_i}(C_{A_i})$ (see Equation (3.5)). The utility function $U_{A_i}^{T^{make}}(C_x)$ is based on $T_{A_i}^{finish}(C_{A_i})$. As shown in Equation (5.10) the utility function $U_{A_i}^{T^{make}}(C_x)$ is calculated by subtracting the time application A_i requires to complete computation when using the cores in C_x from the time required when not using the cores in C_x. Depending on whether the set of cores C_x is already allocated to application A_i (i.e. $C_x \subseteq C_{A_i}$), C_x is imaginary removed from/added to C_{A_i}.

$$T_{A_1,A_2,\ldots,A_M}^{make} = \max_{i \in \{1,\ldots,M\}} T_{A_i}^{finish}(C_{A_i}) \tag{5.6}$$

$$\textbf{Optizimation Goal: } \text{minimize } T_{A_1,A_2,\ldots,A_M}^{make} \tag{5.7}$$

$$\textbf{Application Mapping Quality: } Q_{A_i}^{T^{make}}(C_{A_i}) = \frac{1}{T_{A_i}^{finish}(C_{A_i})} \tag{5.8}$$

$$\textbf{Overall Mapping Quality: } Q^{T^{make}}(A_1, A_2, \ldots, A_M) = \frac{1}{T_{A_1,A_2,\ldots,A_M}^{make}} \tag{5.9}$$

$$U_{A_i}^{T^{make}}(C_x) = \begin{cases} T_{A_i}^{finish}(C_{A_i} \setminus C_x) - T_{A_i}^{finish}(C_{A_i}) & \text{if } C_x \subseteq C_{A_i} \\ T_{A_i}^{finish}(C_{A_i}) - T_{A_i}^{finish}(C_{A_i} \cup C_x) & \text{if } C_x \not\subseteq C_{A_i} \end{cases} \tag{5.10}$$

5.3. Centralized Resource Management

With a given optimization goal (Section 5.2), a performance estimation function (Chapter 4) and constraints (Section 1.2), resource management basically means to solve the resulting optimization problem. Due to the NP-completeness of the problem [CGJ78] and the factorial growth of the search space [MMCM07], it is not possible to solve the problem optimally at runtime for large many-core systems. Therefore, heuristics are required. This Section presents two heuristics for resource management. Both always aim at the globally best solution – potentially resulting in big changes in the mapping whenever an application begins a new phase with different computational demands, or an additional application is executed on the system, or an application terminates.

5.3.1. Hill-climbing Optimization

The first heuristic is a hill-climbing optimization that achieves the resource allocation by greedily (re-)allocating the cores in $\mathcal{C}$ among the applications $\{A_1, A_2, \ldots, A_M\}$ as long as this (re-)allocation improves the overall mapping quality $Q(A_1, A_2, \ldots, A_M)$ determined by the respective optimization goal. The respective commented pseudo-code is presented in Listing 5. In contrast to a simple greedy allocation that allocate each core only once to the momentary best application, the resulting mappings consider the dependency of the application performance and the spatial distribution of the cores (see Section 4.4) by repeatedly reallocating cores until no improvement in $Q(A_1, A_2, \ldots, A_M)$ is achieved.

5.3.2. Iterative Optimization

The second heuristic is actually a combination of the heuristics presented in [SLS07] which is used to decide the number of cores to allocate to each application, and the MAMS heuristic presented in [YGSP13] which is used to actually map the applications to the many-core system (see Section 5.1.2). The first heuristic has been chosen because it produces competitive, near-to-optimal resource allocations for malleable applications for a broad range of input data without the need for fine-tuning [SLS07]. It is an iterative greedy selection scheme that achieves efficient resource utilization, as cores are not wasted on poorly scalable applications. Listing 6 shows the respective pseudo code, adapted from [SLS07].

Figure 5.2.: Combination of [SLS07] and [YGSP13] to achieve an application mapping

Listing 6 itself does not create mapping decisions. A mapping heuristic algorithm like MAMS [YGSP13] needs to be applied to achieve an actual mapping of the applications to the cores on the chip – Figure 5.2 shows the resulting application mapping flow which is used in this thesis. First Listing 6 is used to decide on the number of cores that should be allocated to each application. Then, the MAMS application mapping method [YGSP13] is used to map the applications to cores. As this might lead to cores not allocated to any application, these remaining cores are greedily allocated to the applications.

Listing 5: Centralized Hill-climbing Application Mapping

Input: Initial application mapping $\{C_{A_1}, C_{A_2}, \ldots, C_{A_M}\}$ for a set of applications $\{A_1, A_2, \ldots, A_M\}$, and the respective utility functions $U_{A_i}(\cdot)$
Output: Updated application mapping $\{C_{A_1}, C_{A_2}, \ldots, C_{A_M}\}$

```
gain ← ∞;
while gain > 0 do
    rating_before ← Q(A_1, A_2, ..., A_M);
    // Analyze for each core in the system, whether it would be
       better to (re-)assign it to a different application
    foreach c ∈ C do
        A_max ← —;
        utility_max ← 0;
        // Select application A_max with the highest utility for c
        foreach A_i ∈ {A_1, A_2, ..., A_M} do
            utility ← U_Ai({c});                          // see Section 5.2
            if utility > utility_max then
                utility_max ← utility;
                A_max ← A_i;
            end
        end
        // Select application A_j that c is momentary allocated to
        A_j ← {A_1, A_2, ..., A_M} | c ∈ C_Aj;
        if A_j ≠ A_max then
            // Reallocate the core from A_j to A_max
            C_Aj ← C_Aj \ {c};
            C_Amax ← C_Amax ∪ {c};
        end
    end
    rating_after ← Q(A_1, A_2, ..., A_M);
    gain ← rating_after − rating_before;
    // if gain > 0 then the (re-)allocation resulted in an better
       fulfillment of the optimization goal and another
       optimization is attempted
end
return {C_A1, C_A2, ..., C_AM};
```

Listing 6: Iterative Greedy Optimization Algorithm [SLS07]

Input: Set of applications $\{A_1, A_2, \ldots, A_M\}$ and the respective utility functions $U^{\#}_{A_i}(\cdot)$
Output: Suitable number of cores $|C_{A_i}|$ to allocate to each application

foreach *application* A_i **do**
  $|C_{A_i}| \leftarrow 1$; `// allocate one core to` A_i
  unmark(A_i);
end
while *unmarked application exists* **and** *cores available* **do**
  Greedily choose application A_j that would benefit the most from an additional core using $U^{\#}_{A_j}(\cdot)$;
  $\left|C_{A_j}\right| \leftarrow \left|C_{A_j}\right| + 1$; `// allocate one additional core to` A_j
  Calculate finishing time of all applications;
  if *overall finishing time did not improve* **then**
    $\left|C_{A_j}\right| \leftarrow \left|C_{A_j}\right| - 1$; `//` A_j `does not improve the total finishing time`
    mark(A_j);
    Recalculate finishing time of all applications;
  end
end
return $\{|C_{A_1}|, |C_{A_2}|, \ldots, |C_{A_M}|\}$;

5.3.3. Latency in Centralized Resource Management

Both presented heuristics solve the optimization problem at runtime. When ignoring the communication overhead (i.e. all application information is available to the resource manager and the mapping decisions are not communicated back to the applications), the decision latencies only consist of the time required to compute the mapping heuristics. To get an idea of these latencies, both heuristics have been implemented and benchmarked on one core of an Intel Core i5-4550 CPU. The average resulting computation latency of both heuristics for various combinations of system sizes (number of cores N) and numbers of applications M is shown in Figure 5.3. The pure computation latency to determine an application mapping in a 1024 (32x32) core system with e.g. 48 concurrent applications is in the order of seconds. As the mapping has to be calculated whenever the resource demands of applications change, this limits the adaptability of the resource management. Especially for starting new applications, these latencies are significant. Obviously, the CPU used to benchmark the heuristics is more powerful than a typical core embedded in a Many-Core System. Therefore the presented values only show the lack of scalability and are expected to be even higher when executed on a weaker CPU core. Additional communication delays worsen the total resource management latencies. However, the measurements also show that for smaller Many-Core systems, the scalability of the algorithms for lower numbers of cores is sufficient.

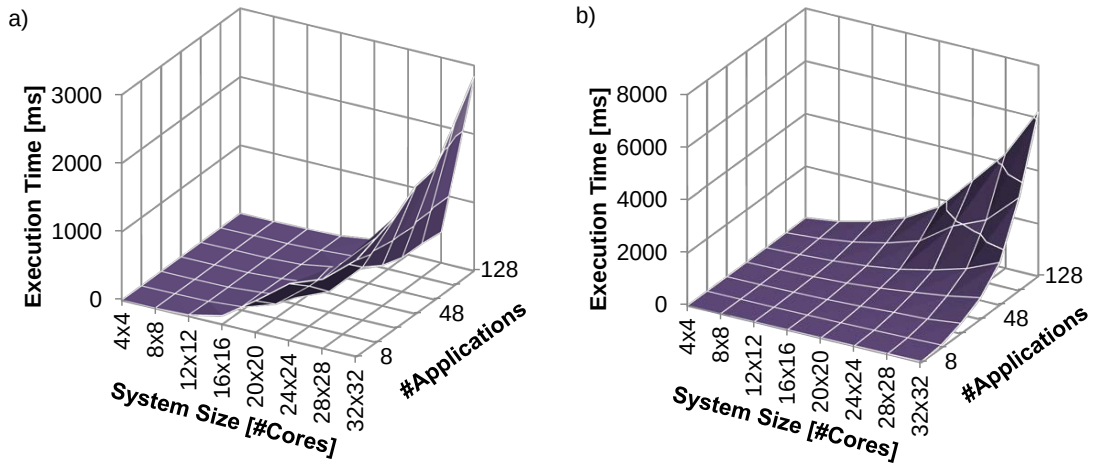

Figure 5.3.: Time required (in ms) to make a mapping decision, measured on an Intel Core i5-4550 for the two presented centralized mapping heuristics: a) Iterative optimization and b) Hill-climbing optimization

5.4. Scalable Distributed Resource Management

As shown in Section 5.3.3, centralized resource management imposes scalability issues when applied to Many-Core systems. A viable approach to achieve scalability is to decouple the optizimation problem from the number of cores N in the system and to parallelize the problem. Therefore, to achieve a high degree of scalability of the resource management, one local resource manager is employed per application for a) managing the cores C_{A_i} of the respective application A_i, and b) communicating the related resource demands to the other resource managers with the goal to achieve a chip-wide coordinated resource management.

The resource management is achieved by a Multi-Agent-System, as introduced in Section 2.3. Compared to centralized resource management, this reduces the computation complexity of resource management, as per mapping decision, only a subset of the system resources and applications is considered which reduces the decision latencies significantly. Agents do not demand system-wide synchronization or knowledge of the global system state. Interaction and communication takes place locally to avoid communication bottlenecks to ensure scalability.

In the scope of this thesis two different strategies for Agent based resource management and a combination of both strategies to an adaptive strategy have been developed:

- The DistRM strategy [KBL+11] is presented in Section 5.4.3
- The low-effort strategy optimized for fine-tuning an existing application mapping is presented in Section 5.4.5
- The adaptive strategy selection AStra that dynamically selects which strategy to use is presented in Section 5.4.6

In all cases, the immediate goal of the Agents interactions is the pairwise re-allocation of cores to/from applications A_i and A_j to optimize the *combined* mapping quality of both applications with the adjusted sets of cores C'_{A_i} and C'_{A_i} is higher than the mapping quality C_{A_i} and C_{A_j} with the sets of cores before the re-allocation, as shown in Equation (5.11). The targeted optimization goal is defined by $Q_{A_i}(C_{A_i})$, as shown in Section 5.2. For system-wide coordinated allocation of cores, Agents are selfless and cooperative, i.e. they strictly follow the global optimization goal.

$$Q_{A_i}\left(C'_{A_i}\right) + Q_{A_j}\left(C'_{A_j}\right) \geq Q_{A_i}\left(C_{A_i}\right) + Q_{A_j}\left(C_{A_j}\right) \tag{5.11}$$

Agents repeatedly and continuously perform application mapping optimizations. This follows the idea of gossip protocols as presented in Section 2.3.2, i.e. the continuous optimization of the allocation of subsets of cores to applications eventually converges the global state to an optimal solution almost surely [RNV09, WZT13].

As the delay between individual optimization trials has influence on the application mapping quality as well as the resource management overhead (see Section 6.2.2), AStra (presented in Section 5.4.6) uses an adaptive delay to automatically achieve a good trade-off.

5.4.1. Multi-Agent System Infrastructure

To implement the Multi-Agent-System, some infrastructure is necessary. It is carefully designed in a way that it does not hamper the scalability and in that it keeps the overhead low.

Proxy Agents

One part of the infrastructure is executed on each core and acts as a proxy for the Agent that momentarily allocated the core. The proxy Agent knows where the actual Agent is located, i.e. the core where the Agent code runs and where the Agent's data structures reside. Note that the Agent responsible for an application that executes a task on core c_i might actually execute on another core c_j. The proxy infrastructure on core c_i is able to forward messages sent by other Agents to its Agent on core c_j. This mechanism allows communicating with the Agent that allocated a certain core without knowing the Agent's location. When the set of cores allocated to A_i changes, only the infrastructure information at the respective cores in C_{A_i} is updated.

Distributed Directory Service

A distributed directory service contains the information, which Agents have acquired cores within a certain region on the chip. One of these directories is instantiated per region of 25 cores each in a 5x5 core grid, as sketched in Figure 5.4. Whenever an Agent initially allocates one of the cores within the region to its application, it registers itself at the corresponding directory. When it releases the last core from the region, it updates the directory accordingly. The resulting protocol state machine is shown in Figure 5.5. The directory does not contain information on which cores or how many cores the Agents manage within the region. Directories do not exchange or synchronize information. These limitations support the scalability, as all information is kept local at any point in time. Due to the reduced amount of information stored in the directory, updating causes only negligible overhead.

Management of Idle Resources

Another part of the infrastructure is a special kind of Agents that manage the idle cores C_{idle} of the system, denoted as *Idle-Agents*. At system initialization, all N cores

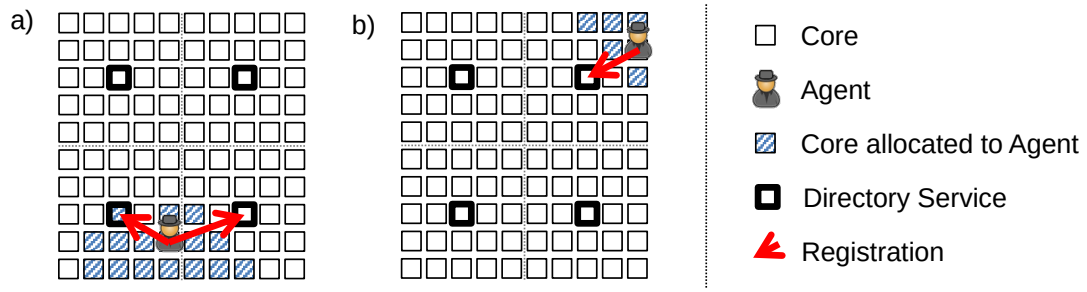

Figure 5.4.: Sketch on how the Agents register themselves at the distributed directory service and on how the directory service is spread throughout the Many-Core system

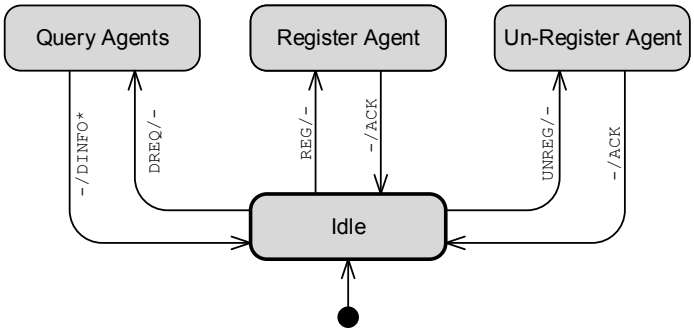

Figure 5.5.: Protocol State Machine used for the distributed directory service. There is no synchronization between directories

Message Definitions

`REG`	Registers an Agent at the directory. Registration takes place as soon as the first core inside the directory area is allocated to the Agent
`UNREG`	Removes an Agent registration from the directory service. Un-registration takes place when the Agent released the last core inside the directory area
`DREQ`	Requests information from the directory
`DINFO`	Message containing information on which Agents are registered at the directory. It does not contain information on which cores the respective Agents have allocated, as this information is not stored in the directory
`ACK`	Generic acknowledgment

in $\mathcal{C}$ are distributed among the Idle-Agents. One Idle-Agent is instantiated per 25 cores (5x5 core grid) to reduce the number of cores each Idle-Agent initially manages. Whenever an application terminates, the applications Agent turns into an Idle-Agent. Similarly, when an Idle-Agent has no more cores to manage, it terminates. The concept of Idle-Agents allows to use the presented protocols without a special case for idle cores – there is **always** a responsible Agent for any core in the system.

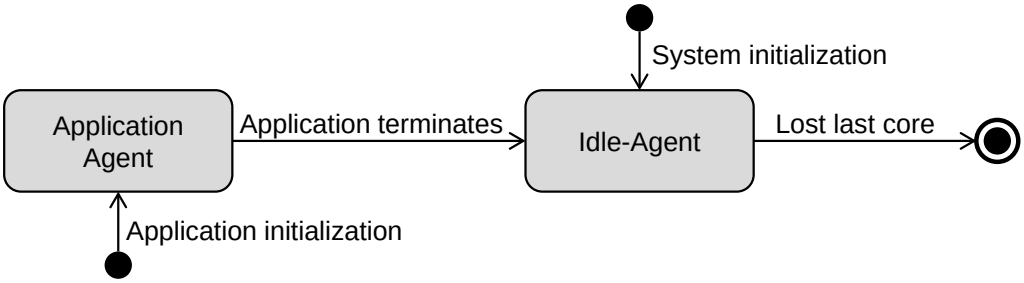

Figure 5.6.: State transitions of an (Idle-)Agent

5.4.2. Application Interface

Each Agent manages the set C_{A_a} of the cores that it has allocated to its application A_a. Each Agent is able to evaluate the utility $U_{A_i}(C_{A_i})$ of the cores C_{A_a} (and additional/different cores) momentarily allocated to its application A_a for the momentary phase P_j because it knows the parameters of the application performance model S_{A_i,P_j}, and the best suited number of cores $N_{A_a,P_j}^{\Phi_{max}}$ of the current phase. The Agent is triggered by its application A_a by an `APP_REQ` signal whenever these parameters change, i.e. when a new phase is about to begin. The Agent informs its application about the availability of additional cores and cores that must no longer be used, e.g. when they got reserved for another application. The application sends a `core available` signal as soon as the task currently mapped to such a core has completed execution. Whenever the set of cores C_{A_a} changes, the Agent informs its application, such that the application can use additional or different cores. Figure 5.7 depicts the interface between an application and its Agent.

The application marks cores as '*used*', as soon as a task is mapped to it. Whenever the interaction between two Agents results in a re-allocation of cores that are marked as '*used*' from application A_a to another application A_b, these cores are marked as '*reserved*'. As soon as the tasks of A_a that use the reserved cores finish execution, the cores are removed from C_{A_a} and the Agent of A_a informs the Agent of A_b regarding their availability to complete the re-allocation. Listing 7 shows the respective pseudo codes.

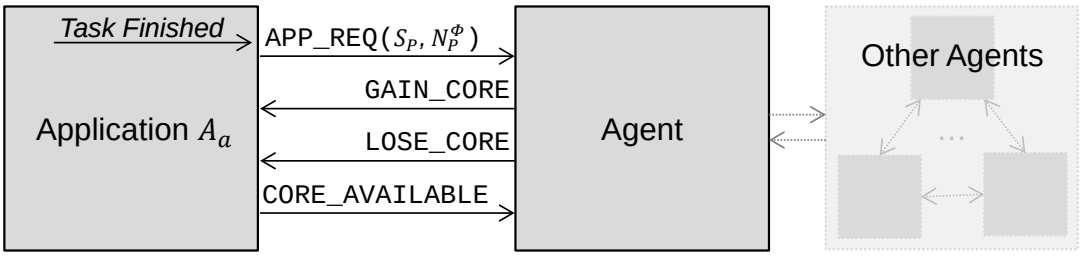

Figure 5.7.: Interaction of Applications and their Agents

Listing 7: Application - Agent Interface

```
SignalApplicationGainCore(c_g):
    if new Task T ready then
        flags[c_g].set(used);
        StartTask(T on c_g);                    // reconfigure application
    end
.....
SignalApplicationLoseCore(c_l):
    if flags[c_l].isSet(used) then
        flags[c_l].set(reserved);
        SignalTaskTermination(T on c_l);        // reconfigure application
    end
.....
SignalTaskFinished(c_f):
    flags[c_f].clear(used);
    if flags[c_f].isSet(reserved) then
        SignalAgentCoreAvailable(c_f);
        flags[c_f].clear(reserved);
    else
        if new Task T ready then
            flags[c_f].set(used);
            StartTask(T on c_f);                // reconfigure application
        end
    end
```

5.4.3. Strategy for Coarse Changes - DistRM

On top of the presented infrastructures, the Agents use different strategies respectively protocols to perform the resource management. The first presented strategy for coarse changes – DistRM [KBL+11] – is used by the resource management Agents to send requests for core re-allocations to larger *regions* C_{req} on the chip as shown in Figure 5.8. A region of size r is a set of cores C_{req} that is defined as core c_c and all cores that are within a Manhattan distance[1] of r to core c_c.

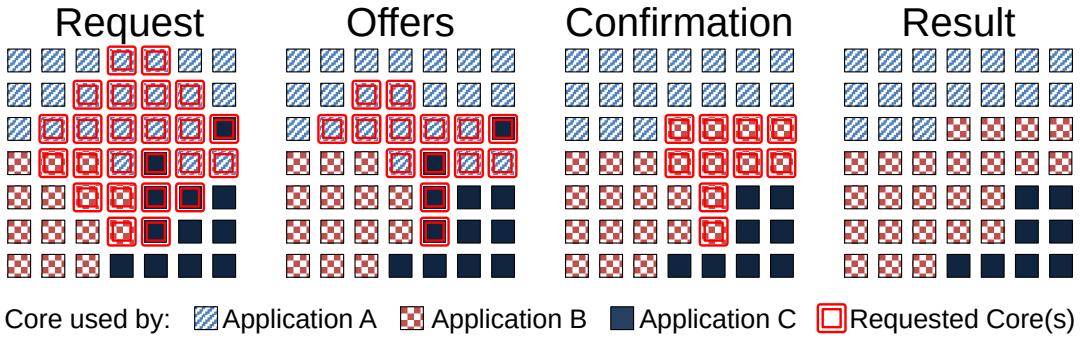

Figure 5.8.: Sketch on the DistRM strategy: Application B's Agent requests a region, collects replies from affected Agents, and decides which cores to re-allocate to its application

Each request contains information about the current mapping $C_{A_{req}}$ of the requesting Agent's application A_{req} as well as the parameters of the speedup function (see Chapter 4) of the momentary application phase P_j. All Agents that already allocated some of the cores in that region to their application receive the request through the Agent System infrastructure (see Section 5.4.1) and evaluate, whether or not the cores in C_{req} are more beneficial to the requesting application A_{req} than for the own application. The requesting Agent collects the replies that contain the information about which cores $C_{offered}$ could potentially be re-allocated to its application A_{req} and then chooses among all these cores which ones to finally re-allocate. A confirmation is sent to each participating Agent and thus, the local application mapping quality for the requested region is improved.

Multiple of these local optimizations may take place in parallel, leading to a fast (re-)allocation of cores to applications. The re-allocation latency (but also the computational complexity for generating the replies and for selecting cores from multiple replies) grow quadratically with the number of cores in the requested regions C_{req}. However, the computational complexity is independent of the size of the many-core system, which makes it scalable for future systems. The fast re-allocation of multiple cores between applications makes this strategy suitable to coarse changes

[1]The Manhattan distance equals the sum of the horizontal and vertical hops used to get from one point to another in a mesh network, see Equation (3.2).

in resource demands (e.g. initial application mapping as shown in Section 5.4.4 or executing a highly parallel phase after an almost sequential one that resulted in an allocation of only a few cores to the application).

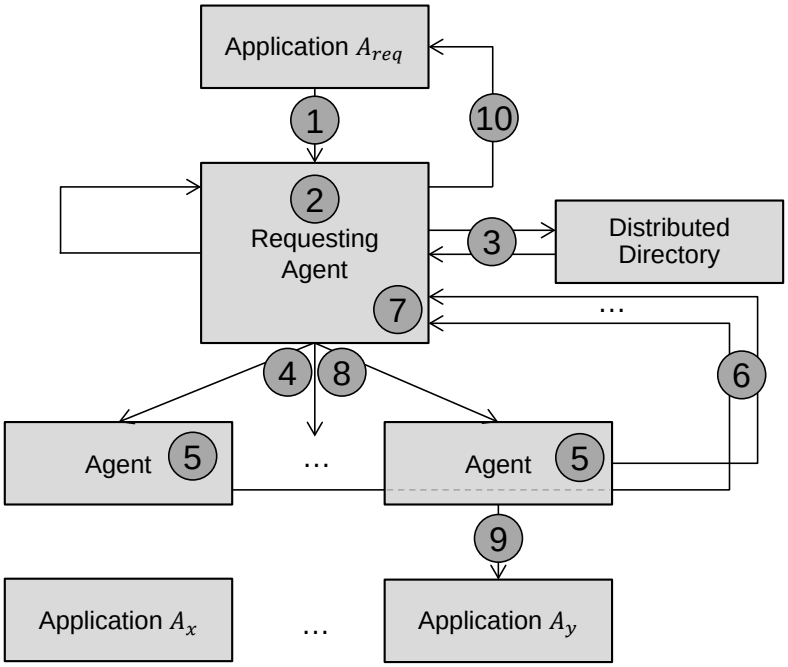

Figure 5.9.: Interaction of the components of the resource management while requesting new cores for application A_{req}

Figure 5.9 presents a flow through the individual components of the Agent System when using the DistRM strategy. In the first step ① the application A_{req} informs its Agent about changing resource demands by sending an `APP_REQ` signal. The Agent then chooses regions on the chip to allocate cores there ②, as detailed later. In each trial, a limited number of regions on the chip are requested in parallel. Therefore, the Agent of A_{req} looks up those Agents that momentarily manage cores in that region ③ and then requests these Agents ④ to evaluate which of the cores allocated to their applications could be allocated to the application of the requesting Agent ⑤ (see Section 5.4.3). They send their offers C_{offer} (containing the parameters that describe the speedup function (see Chapter 4) of their momentary application phase) back to the requesting Agent ⑥. Multiple offers are evaluated by the requesting Agent ⑦. All cores $C_{selected} \subseteq C_{offer}$ that help increasing the overall application mapping quality are allocated to the requesting application A_{req}, as detailed later. Note that not all offered cores are chosen, e.g. if the Agent has received multiple offers or its own application's situation has changed. The Agent then informs offering Agents, whether or not their offer had been selected for re-allocation of cores between the applications and which cores are affected ⑧. The Agents that were selected to release cores inform their own applications to reconfigure and resize ⑨ while in parallel the requesting application A_{req} is informed ⑩ and reconfigures itself for its

new set of cores $C_{A_{req}} \cup C_{selected}$.

A more general description of the DistRM strategy is given by its protocol state machine shown in Figure 5.10. It specifies the legal usage and sequence of the different messages and signals.

DistRM: Request Region Selection

The Agent first examines regions C_{req} spatially close to its own position on core c_a, but with each iteration, the probability to use more distant regions increases. The search is performed in a loop until at least one core is offered to the requesting application A_{req}. As the Agent is not aware of the global system state, it randomly selects $C_{req_1}, \ldots, C_{req_m}$ as $PotentialRegions$ on the chip and aims allocating cores. Up to MAX_PAR_REQS requests are performed in parallel to speed up the search. Therefore, first the Agent randomly selects multiple potential regions on the chip and then performs a pre-selection of regions before initiating the actual requests. To reduce the average communication distance, regions close to the seed core c_a are preferred in the pre-selection process. The longer the Agent (unsuccessfully) tries to allocate cores for its application A_{req}, the higher the probability becomes to use regions that are more distant from the seed core c_a. Therefore, nearby (visualized as 'avoided distance' in Figure 5.11) regions are removed from the list of potential regions with an always-increasing probability, as the Agent most probably examined them before. Note that – with a low probability – not all nearby regions are removed from $PotentialRegions$ to account for changes in the application mapping situation that might result in successful requests in later trials. If more than MAX_PAR_REQS regions remain within $PotentialRegions$, the most distant remaining ones are removed, too. Figure 5.11 shows a sketch on how this leads to an continuous exploration of different regions on the chip around the core c_a. Listing 8 shows the pseudo code that implements the described behavior.

DistRM: Request Handling

A greedy heuristic is deployed to decide which of the cores C_{offer} of application $A_{offerer}$ are offered to A_{req} if the Agent of application $A_{offerer}$ receives a request. The key idea is to re-allocate the cores within a given region to the applications such as the total gain $gain_{total} = gain_{A_{req}} - loss_{A_{offerer}}$ is maximized (see Listing 9). To calculate the respective $gain$ and $loss$, The utility function $U_{A_i}(C_x)$ is used to pursue the global optimization goal as shown in Section 5.2. The selection of cores continues until $gain_{total}$ is maximized, i.e. when the selection of another core c_{req} results in a lower value than the so far achieved $gain_{total}$ All offered cores C_{offer} are reserved for the Agent of the requesting application A_{req}. The reservation is removed after the Agent received the decision, which (if any) of the offered cores are re-allocated to the requesting application A_{req}.

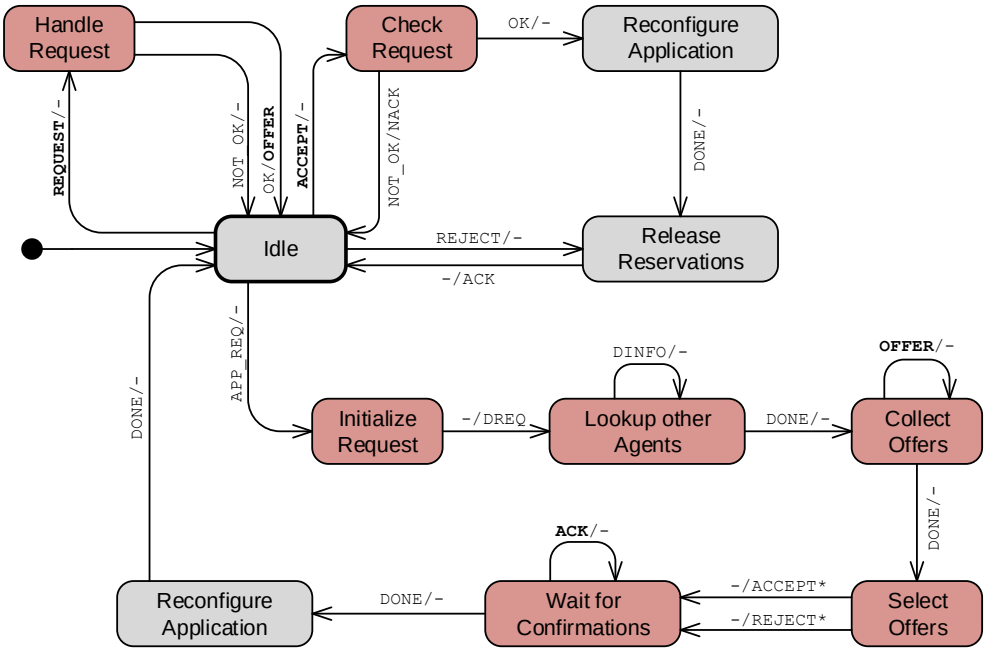

Figure 5.10.: Protocol State Machine for the DistRM Strategy

Message Definitions

APP_REQ	Signal emitted by the application to trigger the resource management to update the resource allocation to the resource demands of the application, e.g. when a new phase is about to begin
REQUEST	Message containing a parametric description of the resource demands of the application (i.e. the parameters of the application performance model) and a description of the resources currently allocated to the application. Receiving Agents reply with an OFFER message
OFFER	Message containing a set of cores $C_{offered}$ that might be transfered to the requesting application together with a parametric description of the respective costs
ACCEPT	Message sent to an offering Agent when offered Resources should be transfered to the requesting application
REJECT	Message sent to an offering Agent when none of the offered Resources should be transfered to the requesting application to allow the release the respective reservations
OK	Generic signal that the action succeeded
NOT OK	Generic signal that the action failed
DONE	Generic signal that the action completed
ACK	Generic acknowledgment
NACK	Generic negative acknowledgment, i.e. the request failed

Listing 8: Selection regions the Agent use to allocate cores in

Input: Core c_a that was chosen as seed core for searching resources for application A_{req}
Output: A set of $Offers$ that contain possible cores to allocate to A_{req}

$tries \leftarrow 0$;
$Offers \leftarrow \emptyset$;
while $Offers = \emptyset$ **and** $tries < MAX\_TRIES$ **do**
  $tries \leftarrow tries + 1$;
  $PotentialRegions \leftarrow selectRandomRegions()$;
  `// Randomized elimination of regions to request resources in. The longer the Agent searches, larger distances are allowed, i.e. with a growing probability nearby regions are removed`
  foreach $C_{req} \in PotentialRegions$ **do**
    if $distance(C_{req}, c_a) < (tries * \frac{MAX\_DIST}{MAX\_TRIES})$ **and** $rand(0,1) > (1/tries)$ **then**
      $PotentialRegions = PotentialRegions - C_{req}$;
    end
  end
  `// Now remove the most distant potential regions`
  while $|PotentialRegions| > MAX\_PAR\_REQS$ **do**
    remove most distant C_{req} from $PotentialRegions$;
  end
  .....
  `// request 'offers' for cores in the remaining regions`
  foreach $C_{req} \in PotentialRegions$ **in parallel do**
    SendMessage `REQUEST`(C_{req});
  end
  wait until timeout or until all `OFFER` messages have been received;
  foreach $Offer \in$ *received* ***`OFFER`*** *message* **do**
    $Offers \leftarrow Offers \cup Offer$;
  end
end
return $Offers$;

Listing 9: Algorithm used by DistRM to handle a request

Input: Incoming request for cores in C_{region} from A_{req}, containing the utility function $U_{A_{req}}(\cdot)$

Output: The set of cores $C_{offered}$ that Application $A_{offerer}$ offers to A_{req}

$C_{offered} \leftarrow \emptyset$;
$gain_{total} \leftarrow 0$;
$gain_{iteration} \leftarrow 0$;
repeat
  $gain_{iteration} \leftarrow 0$;
  foreach *Core* $c_{req} \in \left(\left(C_{A_{offerer}} \setminus C_{A_{offerer}}^{reservations}\right) \cap C_{region}\right) \setminus C_{offered}$ **do**
    $gain_{A_{req}} \leftarrow U_{A_{req}}(C_{offered} \cup c_{req})$; `// see Section 5.2`
    $loss_{A_{offerer}} \leftarrow U_{A_{offerer}}(C_{offered} \cup c_{req})$;
    if $gain_{A_{req}} - loss_{A_{offerer}} > gain_{iteration}$ **then**
      $gain_{iteration} \leftarrow gain_{A_{req}} - loss_{A_{offerer}}$;
      $c_{GreedyChoice} \leftarrow c_{req}$;
    end
  end
  if $gain_{iteration} > gain_{total}$ **then**
    $C_{offered} \leftarrow C_{offered} \cup c_{GreedyChoice}$;
  end
until $gain_{iteration} < gain_{total}$;
$C_{A_{offerer}}^{reservations} \leftarrow C_{A_{offerer}}^{reservations} \cup C_{offered}$; `// reserve until confirmation`
return SendMessage *OFFER*$(A_{req}, C_{offered})$;

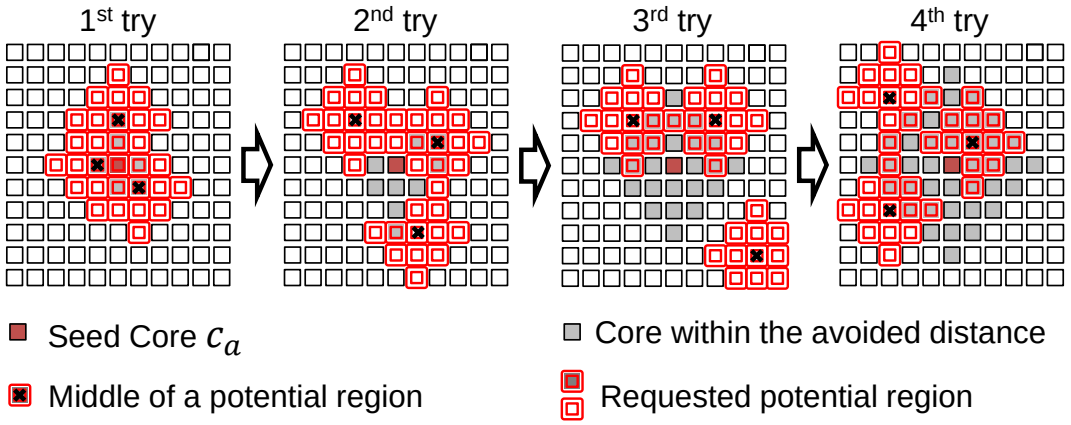

Figure 5.11.: Sketch on how the area around core c_a in which regions C_{req} are probabilistically removed from the potential regions grows with each unsuccessful trial. This leads to an continuous exploration of different regions on the chip around the core c_a while nearby regions are re-requested with a low probability

DistRM: Selection of Cores from Offers

As the $Offers$ were assembled by each Agent without knowing what the other Agents might offer, the superset of the offered cores $C_{offered_1} \cup \ldots \cup C_{offered_m}$ typically contains more cores than required for optimizing the local application mapping. Therefore, after the Agent of application A_{req} received the $Offers$, it needs to select which of the offered cores to allocate to A_{req}. The individual $Offers$ may be accepted fully, partially or not at all. The selection uses a similar algorithm as the generation of theses $Offers$. The Agent greedily picks the cores from the superset of all offered cores that would, if allocated to A_{req}, improve the overall application mapping quality the most until no further gain in application mapping quality could be achieved.

Subsequently, all Agents that offered cores are informed which cores C_{taken_i} have been selected from their offered cores $C_{offered_i}$. They use this information to inform their application about the upcoming loss of theses cores or to release the reservation (or both if only a subset of the cores has been selected). After the offering Agents eventually confirmed the availability of the selected cores, they are allocated to the requesting application A_{req}. Listing 10 shows the respective pseudo code.

Note that each Agent periodically independently requests cores and then selects cores for reallocation from the received $Offers$ (see Section 5.4.6). Therefore, after a certain delay, the Agents that offered cores to A_{req} will receive another request from A_{req}'s Agent that – at that point in time – reflects the situation after the previous $Offers$ have been processed. This way, even without global synchronization, information is disseminated throughout all Agents. This principle

Listing 10: DistRM Selection of cores from $C_{offered}$

Input: The $Offers$ (each containing $C_{offered_i}$) generated by the replying Agents
Output: The set of cores C_{taken} that will be allocated to the requesting application A_{req}

$C_{offers} \leftarrow \bigcup C_{offered_i}$;
$C_{taken} \leftarrow \emptyset$;
repeat
  $gain_{total} \leftarrow 0$;
  foreach *Core* $c_{take} \in C_{offers} \setminus C_{taken}$ **do**
    $gain_{A_{req}} \leftarrow U_{A_{req}}(C_{taken} \cup c_{take})$;
    $loss_{A_{offerer}} \leftarrow U_{A_{offerer}}\left(\left(C_{offered_{offerer}} \cap C_{taken}\right) \cup c_{take}\right)$;
    if $gain_{A_{req}} - loss_{A_{offerer}} > gain_{total}$ **then**
      $gain_{total} \leftarrow gain_{A_{req}} - loss_{A_{offerer}}$;
      $c_{GreedyChoice} \leftarrow c_{take}$;
    end
  end
  if $gain_{total} > 0$ **then**
    $C_{taken} \leftarrow C_{taken} \cup c_{GreedyChoice}$;
  end
until $gain_{total} <= 0$;
foreach $C_{offered_i} \in Offers$ **in parallel do**
  if $C_{taken} \cap C_{offered_i} = \emptyset$ **then**
    `// No core from this offer has been selected, inform offering Agent to release the respective reservation`
    SendMessage `REJECT`(A_i, $C_{offered_i}$);
  else
    SendMessage `ACCEPT`(A_i, $C_{offered_i}$, $C_{taken} \cap C_{offered_i}$);
  end
end
wait until all confirming `ACK` messages have been received;
return C_{taken};

allows DistRM and Astra to scale huge numbers of cores, at the cost of slightly suboptimal decisions that are periodically revisited and improved.

5.4.4. Initialization of New Applications

When a new application A_{req} is about to start running on the system, its Agent is initiated on a random core c_a first. Randomness is applied to achieve load balancing without the need for global system state knowledge. The randomly chosen initial core c_a acts as a seed for searching cores to allocate to A_{req}, however, it might not be part of the set of cores $C_{A_{req}}$ that gets allocated to the application. The DistRM mechanisms (presented in detail in Section 5.4.3) are used to allocate the initial set of cores to the new application A_{req}. After the allocation, the Agents state is migrated from c_a to one of the cores in $C_{A_{req}}$. The flowchart of the application initialization process is shown in Figure 5.12.

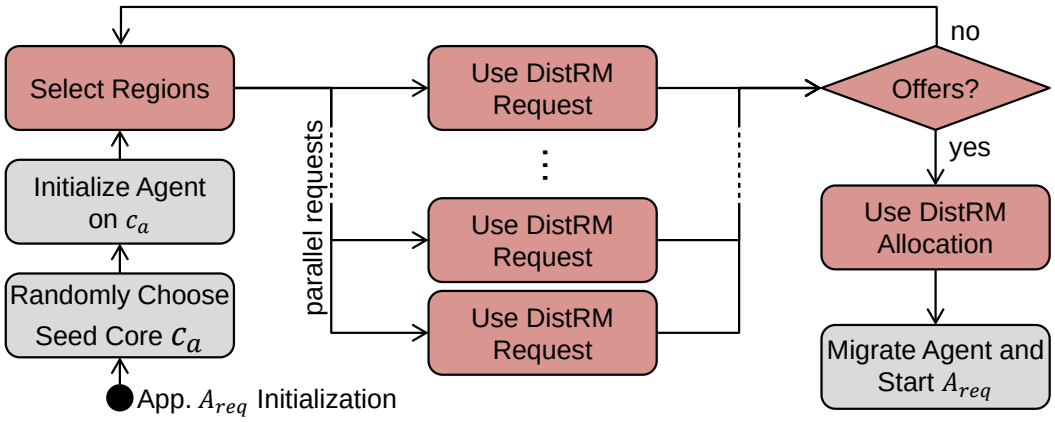

Figure 5.12.: Flowchart followed for initializing a new application A_{req}. The highlighted steps are part of the DistRM strategy and detailed in Listing 8

5.4.5. Low-Effort Strategy for Fine-Tuning

The weakness of the so far presented DistRM strategy is the management of small changes in the application mapping as – in the worst case – for each application optimization request multiple other Agents receive the request and have to reply. This leads to an unreasonable effort for a potentially small gain. Therefore, the proposed low-effort strategy only considers individual cores c_{req} by optimizing the local mapping of two spatially neighboring applications A_{req}, A_{owner} instead of considering whole regions, where (at least before the request) c_{req} is allocated to A_{owner} and A_{req} is searching for additional cores. If A_{req} achieves the higher benefit for core c_{req} (see Section 5.4), its Agent will request to allocate it to A_{req}. Otherwise another core c'_{req} is analyzed. Agents select these cores c_{req} in the boundary $C^{boundary}_{A_{req}}$

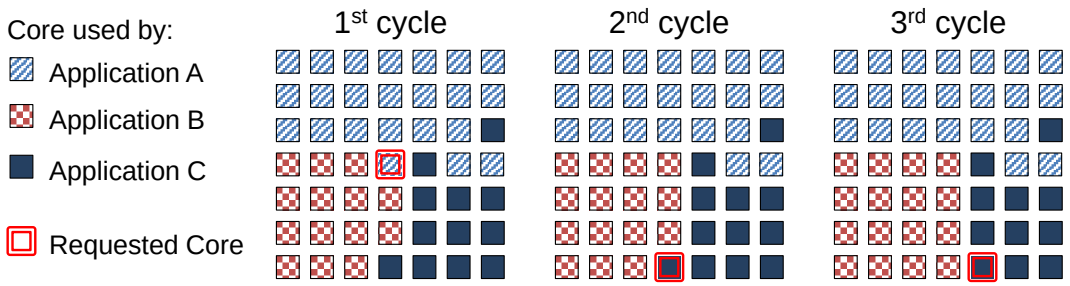

Figure 5.13.: Sketch on how application A_b's Agent uses the low-effort strategy to improve the application mapping

(defined by the topological neighborhood of the cores in $C_{A_{req}}$, see Equation (5.12) and Figure 5.14) of the cores $C_{A_{req}}$ they already manage.

$$C_{A_{req}}^{boundary} = \left\{ c_i \in \left(\mathcal{C} \setminus C_{A_{req}} \right) \mid \exists c_j \in C_{A_{req}}, |c_i^x - c_j^x| + |c_i^y - c_j^y| = 1 \right\} \tag{5.12}$$

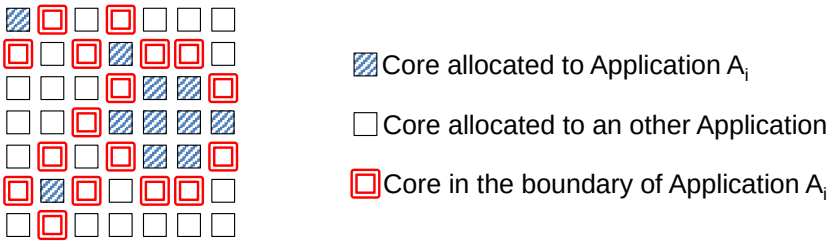

Figure 5.14.: The boundary $C_{A_i}^{boundary}$ of Application A_i

Considering only single cores c_{req} per request significantly reduces the computation complexity and communication. However, to improve the application mapping quality comparable to a request in DistRM (i.e. where the allocation of multiple cores is changed), multiple cycles of the low-effort strategy need to be performed – inducing a higher overhead (many messages sent through the NoC and many requests to be processed). Thus, the low-effort strategy is most-suited for a fine-tuning of the application's mapping after coarse decisions have been made. Figure 5.13 and Figure 5.15 show the optimization process for application mapping.

The algorithm behind the low-effort strategy is split into two parts. The first part shown in Listing 11 is the *requesting* part where each Agent periodically scans the application boundary $C_{A_{req}}^{boundary}$ progressively for new cores. Once a core c_{req} has been chosen, the utility $U_{A_{req}}(c_{req})$ for that core is calculated and – if that core has not been requested by the Agent in the recent past – a `C_REQUEST` message containing the utility of c_{req} to application A_{req} is sent to core c_{req} (Step ① in Figure 5.15). The infrastructure (see Section 5.4.1) running on core c_{req} forwards the message ②

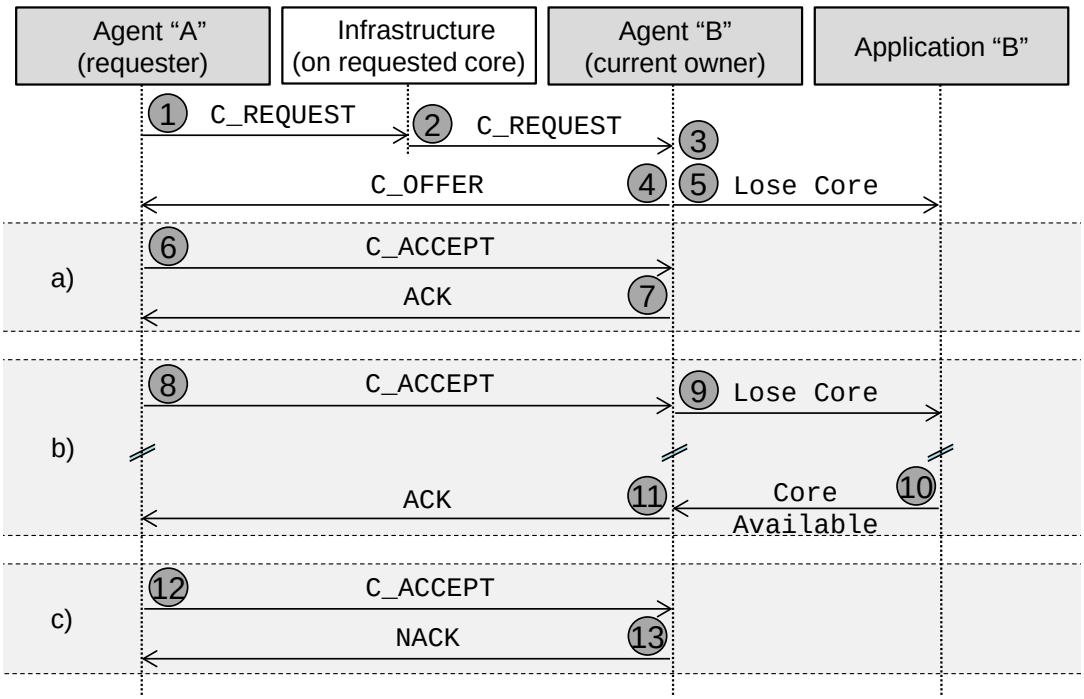

Figure 5.15.: Message-Sequence chart of interactions between Agents in the low-effort strategy

to the Agent that is currently managing core c_{req}. To keep the overhead low, the requesting Agent makes sure, no unnecessary requests are send (i.e. previous attempt failed, so the attempt would fail again if the system state had not changed in the meantime). The Agent does not explicitly wait for a reply to its requests. Instead, a `C_OFFER` message is send to the requesting Agent if the re-allocation of the core would be beneficial. The complete protocol state machine of the low-effort strategy is shown in Figure 5.16.

Listing 12 shows, how an Agent handles an incoming request (Step ③ in Figure 5.15). When its application benefits less from core c_{req} than the rating of the requesting Agent, then the core is re-allocated to the requesting application. When core c_{req} is momentarily not in use by the application, the re-allocation takes place immediately (case 'a)' in Figure 5.15). Otherwise (case 'b)' in Figure 5.15), core c_{req} is reserved for the requesting Agent and the application-local scheduler is informed to release the core (Step ⑤). The reservation for core c_{req} can be overwritten, if in the meantime another Agent with an even higher rating of the core sends a respective request. As soon as the currently executing task finishes execution on core c_{req}, a *core available message* is sent to the Agent which then offers core c_{req} to the requesting Agent with the so far highest request rating – along with the current benefit of the core (Steps ⑥ and ⑦). The offer is confirmed in a three-way handshake if the re-allocation is still beneficial. Requests from previous Agents with a lower request rating are discarded.

Listing 11: Low Effort Strategy: Request Generation

```
// periodically executed to optimize the application mapping
Counter ← 0;     // used to limit the number of trials to find c_req
while Counter < MAX_TRIES do
    Select random core c_req from the application boundary C^boundary_A_req;
    if last_req_time[c_req] ≤ now() − REQ_DELAY then
        RU ← U_A_req (c_req);          // Request Utility, see Section 5.4
        SendMessage C_REQUEST(c_req, RU);    // handled in Listing 12
        last_req_time[c_req] ← now();
    else
        // c_req has recently been requested, better try an other
           core
        Counter ← Counter + 1;
    end
end
```

Listing 12: Low Effort Strategy: Request Handling

Input: Requested Core c_{req}, Requesting Agent RA, Request Utility RU
Output: Message sent to RA, depending on the availability and utility of c_{req}

```
if U_A_offerer (c_req) < RU then
    // Requesting application A_req has a higher utility for c_req
    if flags[c_req].isSet(inuse) then  // reserve core for RequestingAgent
        if not flags[c_req].isSet(reserved) or c_req.resv_utility < RU then
            flags[c_req].set(reserved);
            resv_utility[c_req] ← RU;
            reserved_for[c_req] ← RA;
            SendSignal ApplicationLoseCore(c_req);
            // see Section 5.4.2
        end
    else
        C_A_offerer ← C_A_offerer \ {c_req};
        SendMessage C_OFFER(RA, c_req);
    end
end
```

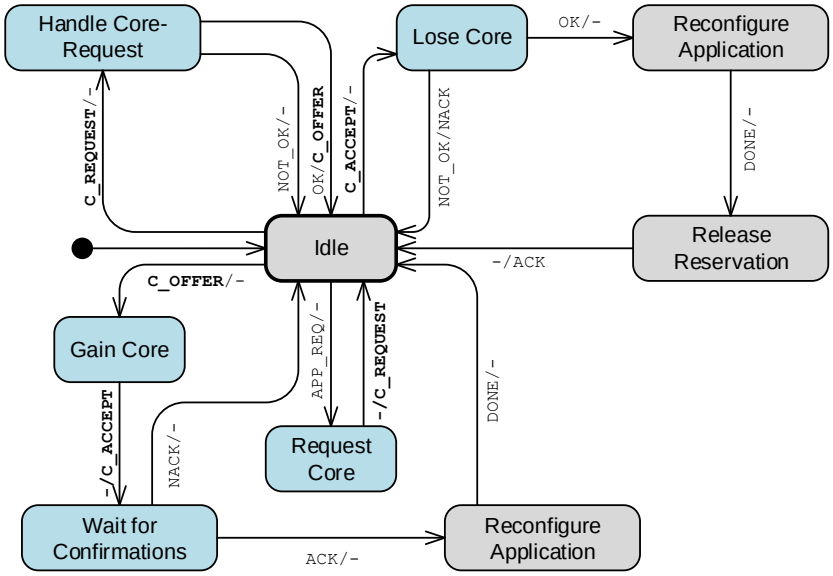

Figure 5.16.: Protocol State Machine for the Low-Effort Strategy

Message Definitions

`C_REQUEST`	Message requesting the transfer of an individual core c_{req} to A_{req}. Contains the evaluation of $U_{A_{req}}(c_{req})$
`C_OFFER`	Message sent in reply to `C_REQUEST` if it is beneficial to allocate the core c_{req} to A_{req}. The core c_{req} is reserved for A_{req} until a request with a higher value of $U_{A_x}(c_{req})$ is received
`C_ACCEPT`	Message that initiates the actual reallocation of c_{req} to A_{req} after it has been offered to A_{req}

5.4.6. Adaptive Strategy Selection - AStra

Agents are able to manage the resources of a Many-Core system by using both of the so far presented strategies. Both optimize the overall application mapping towards the same global goal (see Section 5.2). However, they quite differ in a) the optimization latency and b) the management overhead. DistRM introduces the same overhead for each attempt to optimize the overall application mapping quality – independently of the success of these attempts. The low-effort strategy optimizes the application mapping gradually with a very low overhead. However, the accumulated overhead of multiple gradual improvements compared to a successful optimization attempt of the DistRM strategy is higher.

Therefore, the adaptive strategy AStra combines both strategies. Each Agent individually and adaptively selects which strategy is used to perform the application mapping optimization. The goal is the optimization of the overall application mapping quality while reducing the resource management related overhead. The adaptive selection enables a trade-off between the invested overhead and the expectable gain in application mapping quality. The selection is based on application knowledge about the best-suited number of cores $N_{A_i}^{\Phi_{max}}$ of an application A_i, the number of cores $|C_{A_i}|$ momentarily allocated to application A_i, and the local system state as observed by the previous interactions with other Agents.

Each Agent performs the application mapping independently in a loop. Figure 5.18 gives an overview on how the employed strategy is selected. Figure 5.17 shows the joint protocol state machine.

Within the first step of the optimization loop ②, the Agent checks whether the size of the current set of cores $|C_{A_i}|$ is close to the best-suited number of cores $N_{A_i,P_j}^{\Phi_{max}}$ for the current phase P_j of application A_i. The dynamically adapted weight ω is used to adapt the strategy selection to the system load, i.e. if $\omega = 1$, the Agent will use the DistRM strategy, until $|C_{A_i}| = N_{A_i,P_j}^{\Phi_{max}}$ and then use the low-effort strategy only for fine tuning. The exponential moving average of the values of $|C_{A_i}| / N_{A_i,P_j}^{\Phi_{max}}$ is used to determine ω (see Equation (5.13)). When application A_i starts a new phase ① (i.e. its resource demands change) ω is reset to the momentary value of $|C_{A_i}| = N_{A_i,P_j}^{\Phi_{max}}$. Thus, when the system is under heavy load and the Agent is not able to achieve the best-suited number of cores $N_{A_i,P_j}^{\Phi_{max}}$, ω will decrease. With a smaller ω, the Agent will select the low-effort strategy for larger differences from $|C_{A_i}|$ to $N_{A_i,P_j}^{\Phi_{max}}$ to reduce the generated overhead.

In the next step (③ or ④), the Agent executes the selected strategy. Afterwards ⑤, the Agent a) informs its application A_i about changes in C_{A_i} and b) updates the value of the exponential moving average ω of $|C_{A_i}| / N_{A_i,P_j}^{\Phi_{max}}$ according to Equation (5.13).

$$\omega_n = \frac{1}{10} \cdot \frac{|C_{A_i}|}{N_{A_i,P_j}^{\Phi_{max}}} + \frac{9}{10} \cdot \omega_{n-1} \tag{5.13}$$

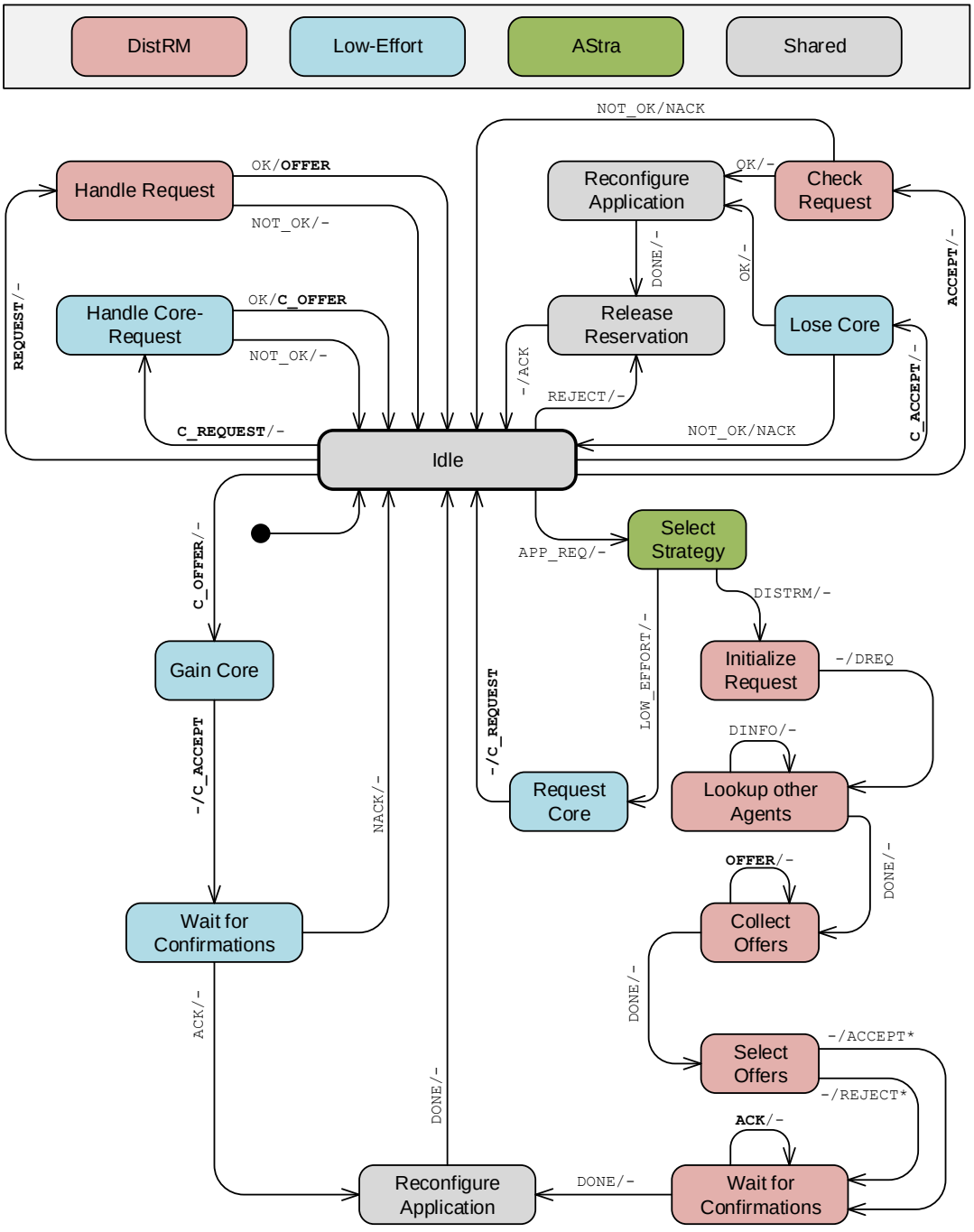

Figure 5.17.: Protocol State Machine including both (Low-Effort and DistRM) Protocols and the Adaptive Strategy Selection (AStra)

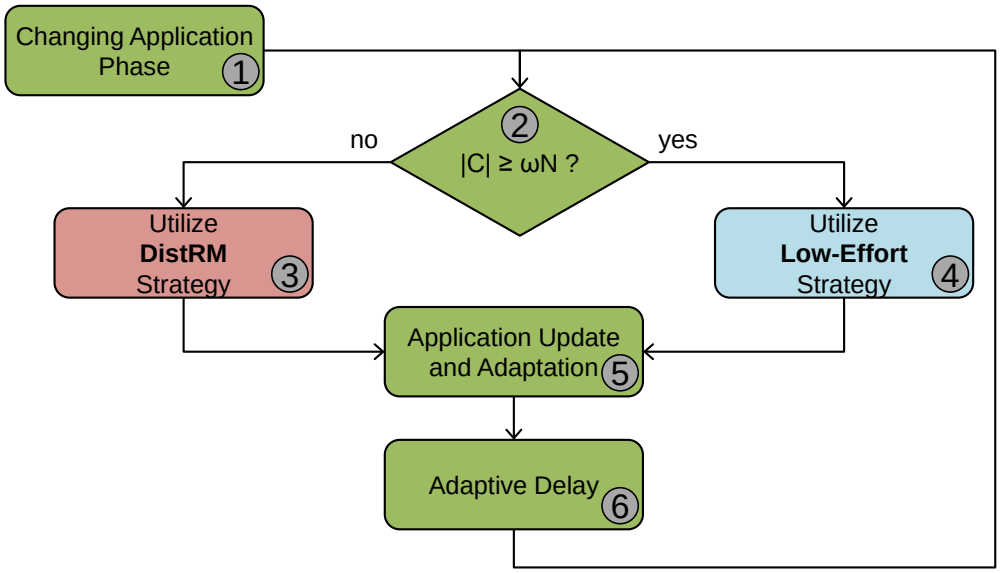

Figure 5.18.: AStra's resource management loop that is executed by each Agent, containing the decision flow followed for deciding, which strategy to use

To reduce the resource management overhead, the optimization loop runs slower (i.e. the delay between two successive optimization attempts is increased) when a) the number of available cores $|C_{A_i}|$ is close to the desired number of cores $N_{A_i}^{\Phi_{max}}$ or b) the last attempts to optimize the application mapping were not successful. As the number of unsuccessful attempts $Failcount$ represents the current system load, it is considered stronger (i.e. squared) in the calculation of the delay to overwrite the decrease of delay caused by missed resource demands in a heavily loaded system. Listing 13 shows details on how the delay is dynamically adapted. The whole optimization loop is triggered whenever the delay expires (6) or the characteristic application properties change (i.e. a new phase is about to begin) (1).

5.5. Summary of Runtime Many-Core Resource Management

Managing the resources of a large on-chip many-core system (a NP-Hard problem) in centrally leads to serious issues in terms of computational complexity and thus high latencies when the number of cores gets too large. The computational complexity is caused by evaluating different potential application mappings based on an application performance model as well as the decision making process itself. This Chapter has shown that the presented distributed resource management for many-core systems is a suitable means to address the scalability issues of centralized resource management with an increasing number of cores. The presented Agent-based distributed resource

Listing 13: Adaptive optimization delay

```
if Last Attempt successfull then
    Failcount ← Failcount * 0.9;                    // exponential falloff
else
    Failcount ← Failcount + 1;
end
Delay ← BASE_DELAY;
if Last used Strategy was Low-Effort Strategy then
    Delay ← Delay/10;
end
Delay ← Delay * (N^{Φmax}_{Ai,Pj} / |C_Ai|) * (1 + Failcount^2);
// The factor N^{Φmax}_{Ai,Pj}/|C_Ai| decreases the delay proportionally to
   the relative demand of additional resources
// Failcount grows if application mapping can not be improved ⇒
   increase delay to avoid potentially unsuccessful requests and
   thus reduce the overhead.  It is used squared to overwrite
   the decrease caused by unmet resource demands in a heavily
   loaded system.
```

management allows distributing the resource-management related computation effort over many cores in parallel. As each Agent only takes care of a reduced subset of the large search space, the computational complexity for each of these parallel working Agents and thus the resource management latency is also significantly smaller. An adaptive strategy selection for distributed runtime resource management was presented which selects the resource management strategy (options: the DistRM strategy for coarse changes, or a low-effort strategy for fine-tuning) adaptively. Chapter 6 will show, that the application mapping quality of distributed approaches is almost as good as it would be with a centralized heuristic even when using a conservative comparison, while significantly improving the scalability.

6. Evaluation and Comparison

This chapter first evaluates the different distributed resource management strategies presented in this thesis (the low-effort strategy in Section 6.1, DistRM in Section 6.2, and AStra in Section 6.3) and then compares them in Section 6.5 to a state-of-the-art distributed resource management approach [ATBS13] (see Section 5.1.3) on various numbers of cores $N = |\mathcal{C}|$ in the many-core system for various application scenarios. To enable the evaluation of resource management on different system configurations, the application scenario needs to fit to the system configuration – e.g. an application scenario that makes use of a 16 core system is not suitable to fully utilize a 1024 core system. Therefore the widely used (e.g. [CB00, Fei03, SCH11, PYL13, ATBS13]) workload generator for many-core resource management evaluation provided by [Dow98] is used. The workload characteristics are based on benchmarks of real workloads on real parallel machines (additional details are given in Appendix B).

However, the generated application scenarios are limited to applications with a single phase, i.e. the applications do not change their scalability at runtime. Therefore, multiple generated application profiles are combined to create larger applications that consist of multiple phases, i.e. each phase in the application model (see Section 3.2) corresponds to one generated application. To keep the amount of work that needs to be performed per application within realistic bounds, the amount of work per phase is divided by the number of phases of the application. At the end, this results in different application scenarios consisting of 8, 16, or 32 concurrent parallel applications with 1, 10, or 100 phases each (e.g., 100 phases represents about 33 iterations of the motivational example consisting of three phases as shown in Section 1.2.1) per system configuration. The achievable parallelism per phase P_j, the best-suited number of cores $N_{A_i,P_j}^{\Phi_{max}}$ per phase P_j, and the amount of work T_{A_i,P_j} (1) to be performed for each application A_i varies according to the boundaries defined by [Dow98]. This means that each application changes its scalability never, ten, or 100 times during its execution to cause low, moderate, or high stress on the resource management. Real-world examples of these application scenarios are high-performance computing with basically no scalability changes during application execution, so-called "Recognition Mining Synthesis" workloads [CCD+08] with an moderate change of scalability with respect to the processed input data, and interactive utilization of the system with continuously changing application phases.

The different distributed resource management strategies are also compared to the centralized resource manager presented in Section 5.3.2 as reference for the mapping quality (see Section 5.2). The centralized manager is a heuristic that provides

considerably good application mappings. However, it is not optimized for many-core systems with hundreds or thousands of cores where the centralized manager induces a high overhead in terms of communication and computation complexity (evaluated in Section 6.4.1 and Section 5.3). In contrast to the presented distributed resource management on many-core systems, it would turn into a bottleneck and limit the dynamic management of the many-core system. In the conservative comparisons, it is assumed that centralized resource management is feasible while ignoring the induced overhead and latencies.

The respective resource managers were evaluated in a system-level simulation environment for many-core systems (see Appendix A). Source-code annotation [BHK+00] was used to determine the computation effort and a NoC simulator to determine the communication effort. The simulations are able to evaluate which core c_i had to perform how many resource management related calculations and how many messages of which size the NoC had to transfer over which distance.

A native bare-metal implementation [BKH14] in the cycle accurate many-core simulator [LRC+11], and an native implementation [PKH12] running as middleware on the Intel SCC (see Section 2.1.2 and Appendix D) were used to validate these estimates. For instance, evaluating the utility function Equation (4.9) for a set of 25 cores takes 2513 cycles in the cycle-accurate measurement whereas the source-code annotation results in 3159 estimated instructions. On average, executing Listing 8 to select the regions to request cores in takes 5313 cycles in the cycle-accurate measurement compared to 4613 estimated instructions. Even though the estimated number of instructions may not be an absolutely comparable number, it is still a very accurate way of comparing the relative overhead of different approaches/algorithms given the same input data.

Please note: Source-code annotation is a more accurate metric to determine the computation effort than the ones used in [KBL+11] and [ATBS13]. Both only counted how often the function to calculate of the utility of cores for an application was executed. However, the computational complexity of that function differs significantly depending on the number of analyzed cores. Therefore, the presented results differ from the ones presented in [KBL+11] and [ATBS13].

The communication effort represents the overall utilization of the communication infrastructure. It is determined by the sum of the length s of all messages multiplied by the distance w each message took through the NoC (see Section 3.1), as shown in Equation (6.1).

$$\text{Communication Effort} = \sum_{i=1}^{\#\text{messages}} w_i * s_i \tag{6.1}$$

To ensure a fair comparison, all approaches were simulated using identical application scenarios and system model configurations (see Chapter 3).

The evaluation focuses on evaluating the resource management strategies and their overhead. AStra is not limited to a specific application and/or hardware platform, it is applicable as long as applications are able to adopt themselves to varying sets of cores, e.g. by using master/slave parallelism and starting/terminating workers depending on the cores available to the application (or more sophisticated means, as described in Section 2.2). Therefore, the evaluation of the resource management strategies does not consider application-internal overheads like scheduling tasks to cores, synchronization, and data transfer, as this overhead depends on the specific application implementation and hardware platform and there is no way to generalize this overhead. Considering the application-internal overhead might result in a lower *absolute* overall application performance. However, the centralized resource manager modifies the set of cores of all applications very frequently, i.e. always when any one of the applications triggers the resource management. Therefore, the *relative* system performance that is used to evaluate the application mapping quality of the distributed resource management strategies compared to centralized resource management might even be higher when considering these details. This results in a conservative evaluation of the application mapping quality of the distributed resource management strategies.

To account for stochastic components in the resource management strategies, each configuration is simulated ten times. The average results of these runs are presented and discussed in the following.

6.1. Low-Effort Strategy

The low-effort strategy for distributed resource management aims to iteratively improve the application mapping while considering only a single core at each optimization trial, see Section 5.4.5. Each Agent executes the strategy in a loop until its application finishes execution. The Agents wait for a certain time between two consecutive optimization attempts to limit the resource management overhead. The influence of this delay between optimization trials and the application mapping quality for 16 concurrently executed applications is shown in Figure 6.1 for an 256 core system and a 1024 core system in Sub-Figures a) and b) respectively. The application mapping quality quantifies how efficient the resource management is able to operate the system. It is determined by the selected optimization goal, as presented in Section 5.2. In the presented results, it is based on the average speedup of the concurrently executing applications (see Section 5.2.1). The centralized resource manager is used as reference for the mapping quality while ignoring its overhead. It is clearly visible that the mapping quality correlates with the delay between the optimization trials. However, the overhead grows significantly for smaller delays. As shown in Section 5.4.6, this insight is exploited in the AStra Strategy by means of an adaptive delay between optimization trials.

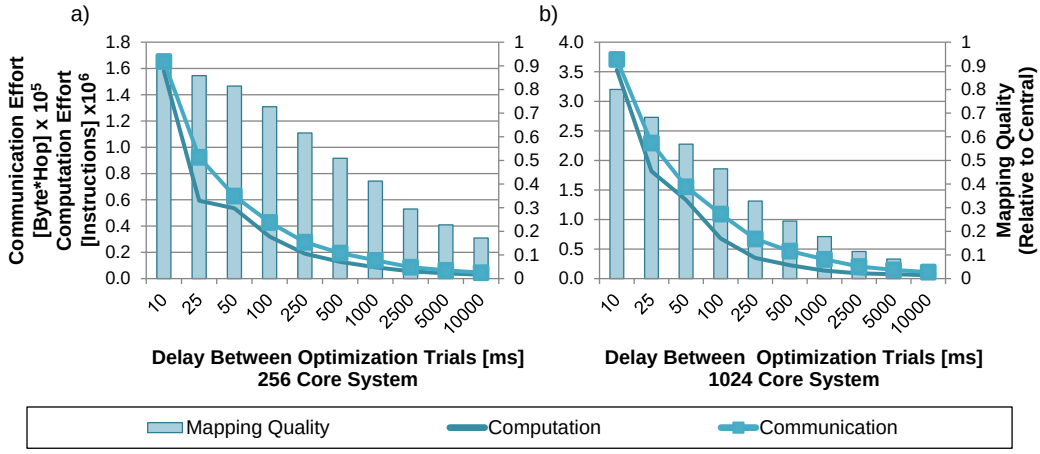

Figure 6.1.: The influence of the delay between optimization trials of the low-effort strategy and the application mapping quality of 16 applications concurrently executed on an 256 core system and on a 1024 core system in Sub-Figures a) and b) respectively. The application mapping quality is given as a relative value of the achieved mapping quality of the potentially infeasible central resource manager while ignoring its overhead.

6.1.1. Scalability Analysis

In Figure 6.2 the scalability of the low-effort strategy is evaluated with respect to the number of concurrently executing applications and the number of cores in the systems. For the presented results, the delay between consecutive optimization trials of each Agent is set to 100ms as it provides a good trade-off between the mapping quality and the induced overhead. However, the relative values scale with the delay as shown in Figure 6.1. Sub-Figures a), b) and c) show the computation effort for various system sizes from 16 to 1024 cores for 8, 16 and 32 concurrent applications. Sub-Figures d), e) and f) show the respective communication effort, and Sub-Figures g), h) and i) the application mapping quality relative to the centralized resource manager while ignoring its overhead. Only the resource management related communication and computation is shown (i.e. the application internal communication is omitted).

Please note: The number of cores in the system N indicated on the x-axis of all following diagrams grows quadratically (i.e. 16, 64, 144, ...). Therefore, a linear relationship of the respective effort and the number of cores in the system looks like quadratic growth and a linear shape of the presented curves actually represents a "better than linear" scalability

Different application scenarios are evaluated: In Sub-Figures a), d) and g) the scalability of the applications stays constant over their whole execution time, i.e. they consist of only one phase (see Section 3.2) and a steady state in which no

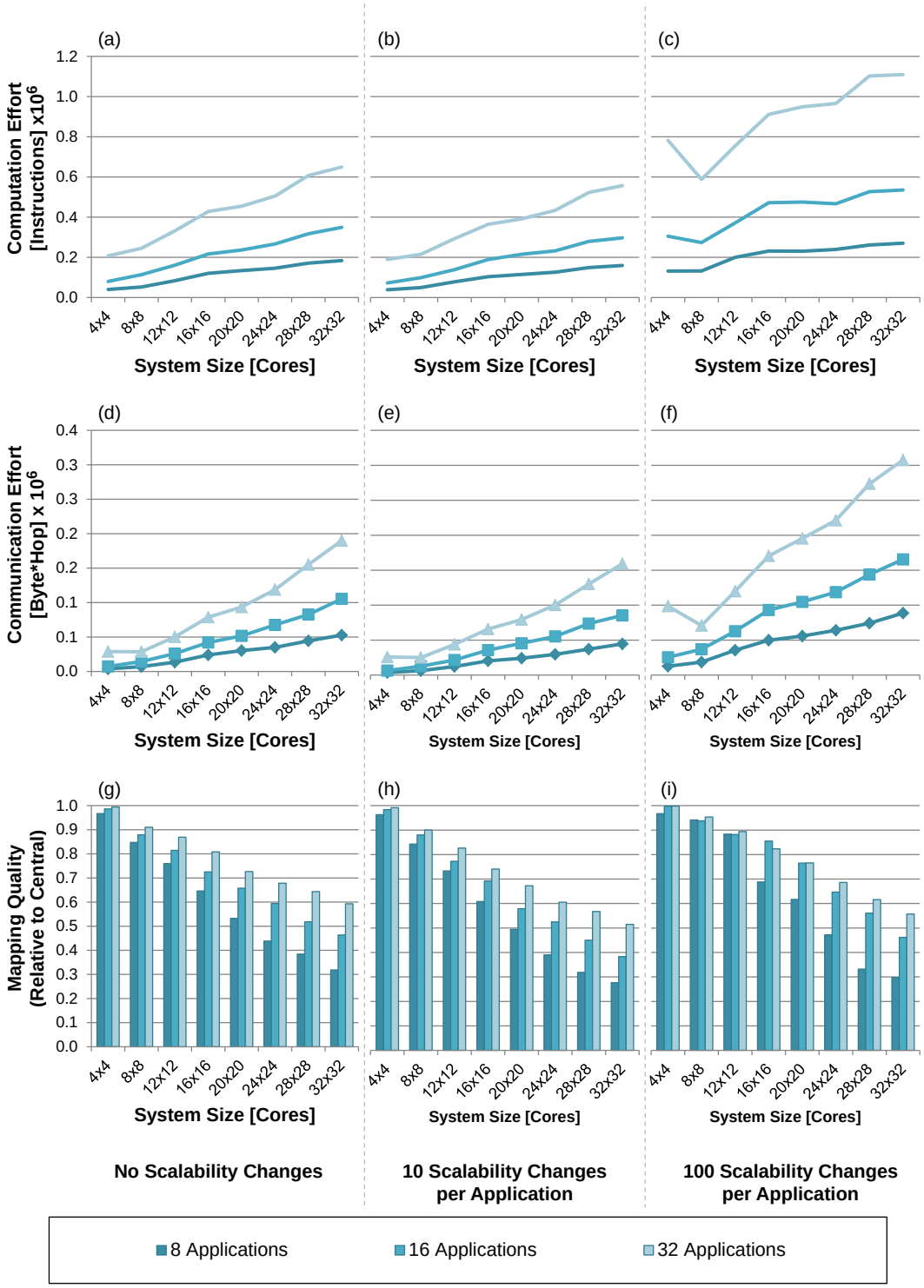

Figure 6.2.: The computation effort (a), b) and c)), communication effort (d), e) and f)) and the relative application mapping quality compared to centralized resource management (g), h) and i)) achieved by the low-effort strategy for various application scenarios and many-core system configurations

further optimization is meaningful is achievable. In Sub-Figures b), e) and h) the applications change their scalability ten times and in Sub-Figures c), f) and u) 100 times respectively, i.e. they represent (highly) dynamic workload scenarios that benefit from a continuous adaptation of the application mapping.

The computation effort as well as the communication effort scale better than linearly with the number of cores in the system and the number of concurrently executing applications. With more cores in the system, on average more cores are allocated to each application A_i and therefore the size of the boundary $\left|C_{A_i}^{boundary}\right|$ (see Section 5.4.5) of each application grows. As Agents do not re-request cores that had been requested recently, the number of cores in the application boundary directly influences the number of requests sent to other Agents and thus on the computation and communication overhead. In case that there are more applications ready for execution than there are cores in the system, the Agents that do not have cores allocated for their application repeatedly send out (unsuccessful) requests which cause the small peak of computation and communication overhead in Sub-Figures c) and d) in the case of 32 applications but only 16 cores in the system.

With the selected delay of 100ms between optimization trials, the low-effort strategy achieves considerably good results on systems with less than about 100 cores. However, it is obvious that the low-effort strategy alone is not suitable for resource management of larger systems as there are too many possible cores $C_{A_i}^{boundary}$ in each applications A_i boundary to evaluate in time. As shown in Sub-Figure 6.1 b), even 10ms between each optimization trial do not result in good application mappings while significantly increasing the induced overhead.

In summary, the low-effort strategy requires a comparably (see Section 6.5) low computation and communication overhead to perform the resource management and is very scalable with respect to the number of applications and the number of cores in the system. However, the achieved application mapping quality degrades significantly for growing system sizes. While it is possible to improve the application mapping quality by means reducing the delay between subsequent optimization trials, this also increases the overhead significantly. An adaptive delay that automatically trades off the induced overhead against the achieved gains in application mapping quality is introduced in the AStra Strategy (Section 5.4.6) and evaluated in Section 6.3.

6.2. DistRM Strategy

The DistRM strategy (explained in detail in Section 5.4.3) also performs the resource management in an loop until its application finishes execution. In each optimization trial, one or multiple requests are sent in parallel to whole regions (i.e. multiple spatially neighboring cores) that affect one or multiple other Agents that momentary have cores allocated in these regions. In Section 6.2.1 the influence of the number

of cores per request and the number of parallel request per optimization trial is analyzed. Section 6.2.2 shows how the delay between two consecutive optimization trials of each Agent influence to overall mapping quality and the induced overhead. To complete the evaluation of the DistRM strategy, it's scalability with respect to the number of concurrently executing applications and the number of cores in the many-core system is analyzed in Section 6.2.3.

6.2.1. Number of Cores per Request and Parallel Requests

The influence of the number of cores considered per optimization request and the number of parallel requests on the computation effort, communication effort and the resulting application mapping quality when using the DistRM strategy on a 1024 core system is shown in Figure 6.3. The respective evaluations on a 256 core system are shown in Figure 6.4.

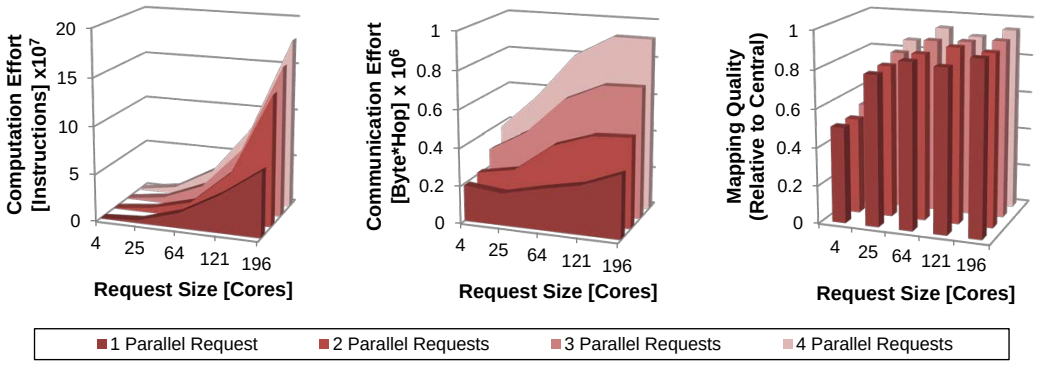

Figure 6.3.: Influence of the number of cores per optimization request and the number of parallel requests on the computation effort, communication effort and the resulting application mapping quality when using the DistRM strategy on a 1024 core system

As expected, the computation effort grows quadratically with the number of cores `REQ_SIZE` in the requested regions. To keep the computation effort and thus the latency of each request manageable, the number of cores per request region should stay rather low. However, too small numbers of cores per region result in comparably bad application mappings. When comparing the application mapping quality achieved with only four cores per request on a 256 core system with the one achieved on a 1024 core system, it is noticeable, that the 1024 core system suffers from a higher degradation than the 256 core system. Therefore, in the following experiments, the number of cores in the request regions is adapted to the number of cores in the many-core system according to Equation (6.2). The resulting values of `REQ_SIZE` for different many-core system configurations are shown in Table 6.1.

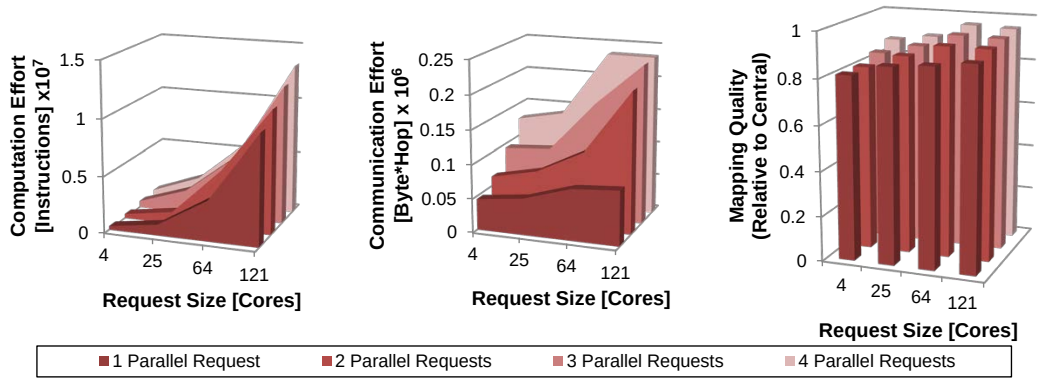

Figure 6.4.: Influence of the number of cores per optimization request and the number of parallel requests on the computation effort, communication effort and the resulting application mapping quality when using the DistRM strategy on a 256 core system

$$\texttt{REQ\_SIZE} = \lceil \sqrt[4]{N} \rceil^2 \tag{6.2}$$

Table 6.1.: Request region size for different many-core system configurations

System Size N	16	64	144	256	400	576	784	1024
REQ_SIZE	4	9	16	16	25	25	36	36

The effect of the number of parallel requests on the computation effort looks surprising at first as there is no huge difference in the computation effort (but also in the application mapping quality) when going from two to three or four requests in parallel. This is caused by the effect that actually too many cores from multiple different Agents are offered (and thus reserved) to a requesting Agent. Not all of these cores are actually selected and a `REJECT` message is send to the offering Agents. Within the time frame between sending the `OFFER` message and receiving the `REJECT` message, these cores are ruled out of evaluation when receiving other requests (see Section 5.4.3). As the biggest fraction of the computation effort is caused by calculating the utility function for requested and/or offered cores, the overall computation effort does not increase further. In contrast, the communication effort increases basically linearly with the number of parallel requests as multiple (potentially unsuccessful) `REQUEST`, `OFFER`, and `REJECT` messages have to be transmitted. As there is also no significant improvement in application mapping quality, the number of parallel requests is set to two in the following evaluations.

6.2.2. Optimization Delay

Just as with the low-effort strategy (evaluated in Section 6.1), the mapping quality of the DistRM strategy increases with shorter delays between the optimization trials. Figure 6.5 shows how the delay between optimization trials influences the induced overhead as well as the resulting application mapping quality when using the DistRM strategy on a 256 core system (Sub-Figure a)) and on a 1024 core system (Sub-Figure b)).

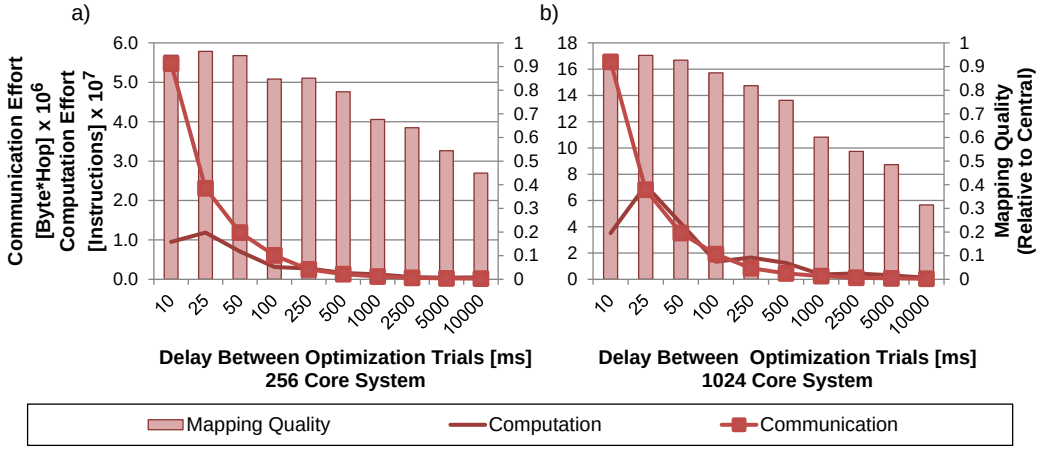

Figure 6.5.: The influence of the delay between optimization trials of the DistRM strategy and the application mapping quality of 16 applications concurrently executed on an 256 core system and on a 1024 core system in Sub-Figures a) and b) respectively. The application mapping quality is given as a relative value of the achieved mapping quality of the potentially infeasible central resource manager while ignoring its overhead.

The correlation between the induced overhead and the resulting application mapping quality when using the DistRM strategy is not as strong as when using the low-effort strategy, i.e. shorter delays increase the overhead significantly without necessarily increasing the application mapping quality. The drop in computation effort in case of 10ms delay between consecutive optimizations compared to a delay of 25ms is caused by requests received by the Agents before previous `OFFER` messages have been replied and thus the resulting reservations have been not been released. As momentary reserved cores are not considered for replying to incoming `REQUEST` message (see Section 5.4.3), the computation effort is actually reduced – similar to the case of too many parallel requests sent out by the Agents evaluated in the previous Section. The communication effort, however, is significantly increased.

Therefore, in the following evaluations, the delay between two consecutive optimization trials of the DistRM strategy is set to 500ms, as this delay provides a good trade-off between application mapping quality and induced overhead. Applications that change their scalability profile at runtime – i.e. applications that consist of

multiple phases, see Section 3.2 – may trigger optimization trials earlier, i.e. whenever a new phase is about to begin or after 500ms have passed since the last optimization trial.

6.2.3. Scalability Analysis

The scalability of the DistRM strategy is analyzed similar to the low-effort strategy in Section 6.1.1. The respective results are shown in Figure 6.6. Again, Sub-Figures a), b) and c) show the computation effort for various system sizes from 16 to 1024 cores for 8, 16 and 32 concurrent applications. Sub-Figures d), e) and f) show the respective communication effort, and Sub-Figures g), h) and i) the application mapping quality relative to the centralized resource manager while ignoring its overhead. Different application scenarios are evaluated: In Sub-Figures a), d) and g) the scalability of the applications stays constant over their whole execution time, i.e. they consist of only one phase (see Section 3.2) and a steady state in which no further optimization is meaningful is achievable. In Sub-Figures b), e) and h) the applications change their scalability ten times and in Sub-Figures c), f) and u) 100 times respectively, i.e. they represent (highly) dynamic workload scenarios that benefit from a continuous adaptation of the application mapping.

The overall computation and communication effort in all scenarios scales linearly with the number of concurrently executing applications and the number of cores in the many-core system. Again, please note the quadratic growth of the number of cores shown on the x-axis. In case of frequent changes in the workload (Sub-Figures c), f), and i)), the induced overhead grows significantly as the time that passes between changes in the scalability is shorter than the selected delay between optimization trials. In case of moderate changes in the workload (Sub-Figures b), e) and h)) there is a slightly increase overhead compared to the case of steady workload (Sub-Figures a), d) and g)) as the time that passes between changes in the scalability is about the same than the delay between the periodic optimization trials.

Using the DistRM strategy with the parameters presented in Section 6.2.1 results in considerably good application mappings on systems with up to 20x20 cores. The drop in application mapping quality for more cores in the system suggests that a more aggressive configuration would result in better application mapping quality for larger systems at the cost of an increased overhead. AStra, presented in Section 5.4.6 and evaluated in Section 6.3 addresses this weakness by means of a dynamic adaptation of the delay between optimization trials to improve the application mapping quality as well as to reduce the induced overhead in case no improvement is achievable.

Compared with the low-effort strategy, DistRM induces a higher computation and communication overhead to perform the resource management. A direct comparison is given in Section 6.5. Still, DistRM is very scalable with respect to the number of applications and the number of cores in the system. The achieved application

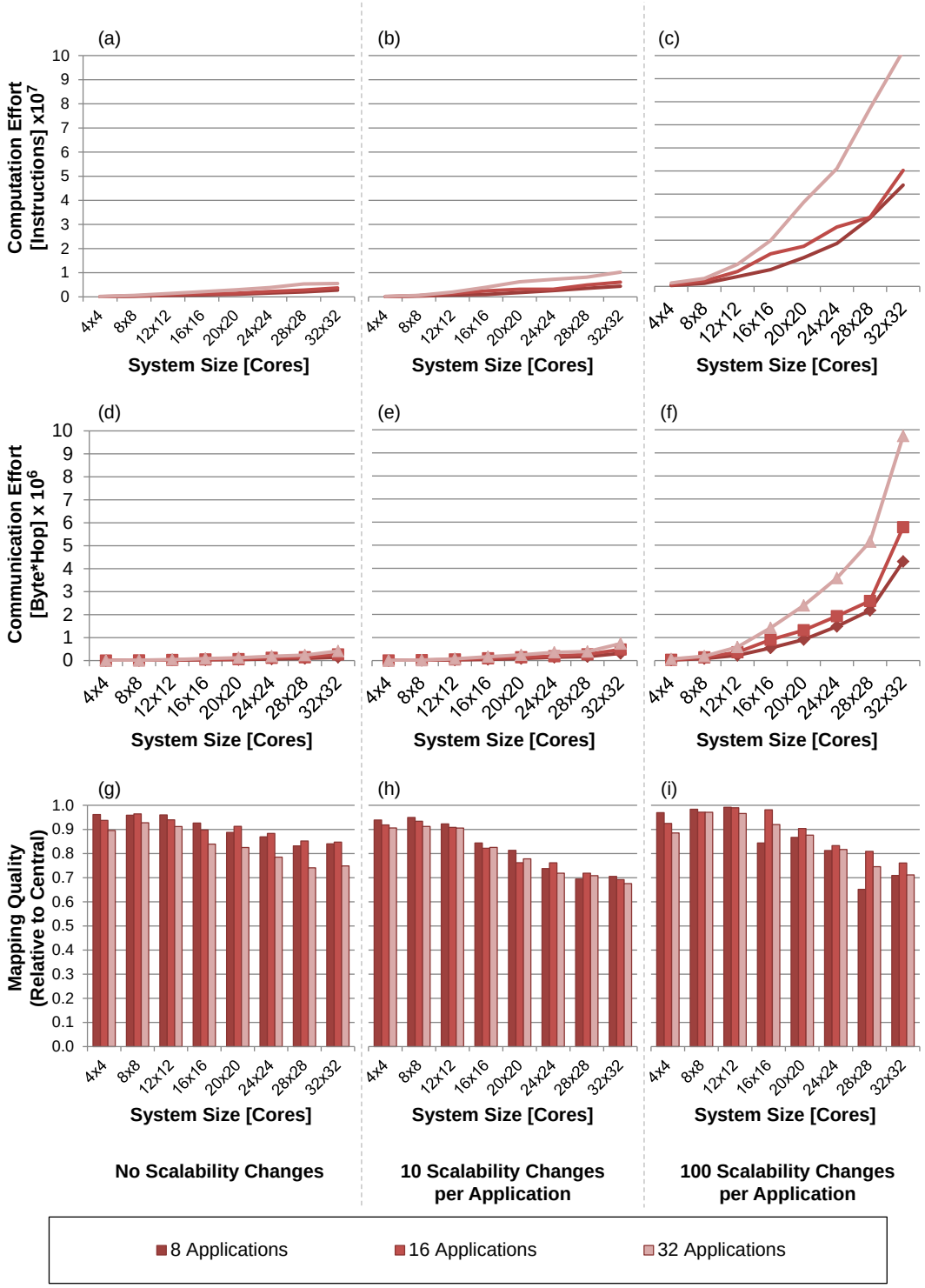

Figure 6.6.: The computation effort (a), b) and c)), communication effort (d), e) and f)) and the relative application mapping quality compared to centralized resource management (g), h) and i)) achieved by DistRM for various application scenarios and many-core system configurations

mapping quality is generally higher than when using the low-effort strategy. With the current configuration, especially on many-core systems with a large number of cores the mapping quality might be improved by means of larger requests as shown in Figure 6.3, or by more frequent optimization trials as shown in Figure 6.5, or both. However, both possibilities would result in an increased overhead. The next Section 6.3 will show, how AStra manages to improve the application mapping quality while at the same time reduces the induced resource management related overhead.

6.3. AStra: Adaptive Strategy Selection

AStra is an adaptive strategy that combines the two so far evaluated strategies (the low-effort strategy and DistRM) to perform the resource management. As shown in Section 5.4.6, the resource management is performed in a loop by each Agent until its application finishes execution. In contrast to the so far evaluated strategies, the delay between consecutive executions of that optimization loop is adapted to the current system state. The influence of the minimal allowed delay between optimization trials is evaluated in Section 6.3.1. AStra adaptively selects which strategy to use for the application mapping optimization. An evaluation of how this selection is performed in different application scenarios and how the adaptive delay is used to reduce the induced overhead is presented in Section 6.3.2. Finally, Section 6.3.3 analyses the scalability of AStra and presents the achieved application mapping quality.

6.3.1. Influence of Optimization Delay

Figure 6.7 shows how the minimal allowed delay between optimization trials influences the induced overhead as well as the resulting application mapping quality when using AStra on a 256 core system (Sub-Figure a)) and on a 1024 core system (Sub-Figure b)). Similar to the so far evaluated resource management strategies, the mapping quality of AStra increases with shorter delays between the optimization trials. In contrast to the other strategies, AStra uses an adaptive delay, i.e. the stated delay is the minimal delay between consecutive optimization trials. This allows to select an lower minimal delay than for using the DistRM strategy without inducing huge amounts of computation and communication overhead. In the following evaluations, a conservative value of 100ms is used. In the following Section 6.3.2 it is shown how this delay is adapted at runtime.

6.3.2. Adaptation Evaluation

The key advantage of AStra is the adaptive selection of the resource management strategy. The basic idea is to select the DistRM strategy to achieve huge changes

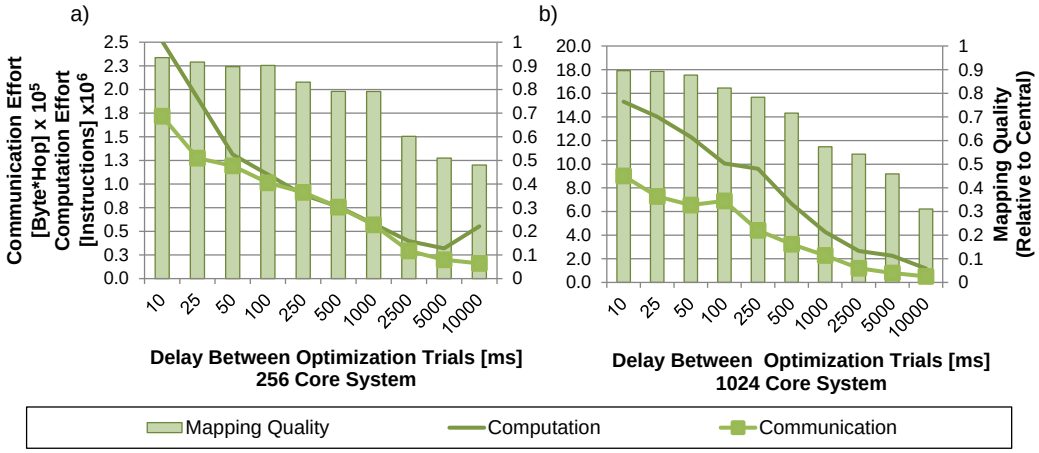

Figure 6.7.: The influence of the minimal allowed delay between optimization trials of the AStra strategy and the application mapping quality of 16 applications concurrently executed on an 256 core system and on a 1024 core system in Sub-Figures a) and b) respectively. The application mapping quality is given as a relative value of the achieved mapping quality of the potentially infeasible central resource manager while ignoring its overhead.

in the application mapping (e.g. when starting a new application) and to use the low-effort strategy otherwise, see Section 5.4.6. Figure 6.8 shows the selection of the respective strategy for different scenarios for applications that do not change their resource demands at runtime. At the start of the simulation, eight applications start execution. Sub-Figure a) shows the number of cores allocated to each application in each time-step. All applications immediately can start execution, after about two seconds, all cores are allocated to applications. At simulation time 10,000ms four additional applications start execution. The re-allocation of cores to these additional applications is performed in several milliseconds. At simulation time 20,000ms, five applications are terminated. The gradual re-allocation of the released resources is clearly visible.

A detailed analysis of the application mapping optimization trials and the number of allocated resources for two selected applications A_A and A_B is given in Sub-Figures b) and c), respectively. Application A_A is highlighted in green in Sub-Figure a), Application A_B is highlighted in red. The vertical lines indicated by color, which resource management strategy was used at which point in time. A red vertical line indicates, that the DistRM strategy was used, a blue vertical line indicates an optimization trial with the low-effort strategy. The distance between red and/or blue vertical lines indicates the adaptive delay between consecutive optimization trials. Grey vertical lines indicate that the applications Agent answered an incoming resource request.

For both applications it is clearly visible how first the DistRM strategy is used to allocate the initial set of cores with a very low latency. Afterwards, mostly the

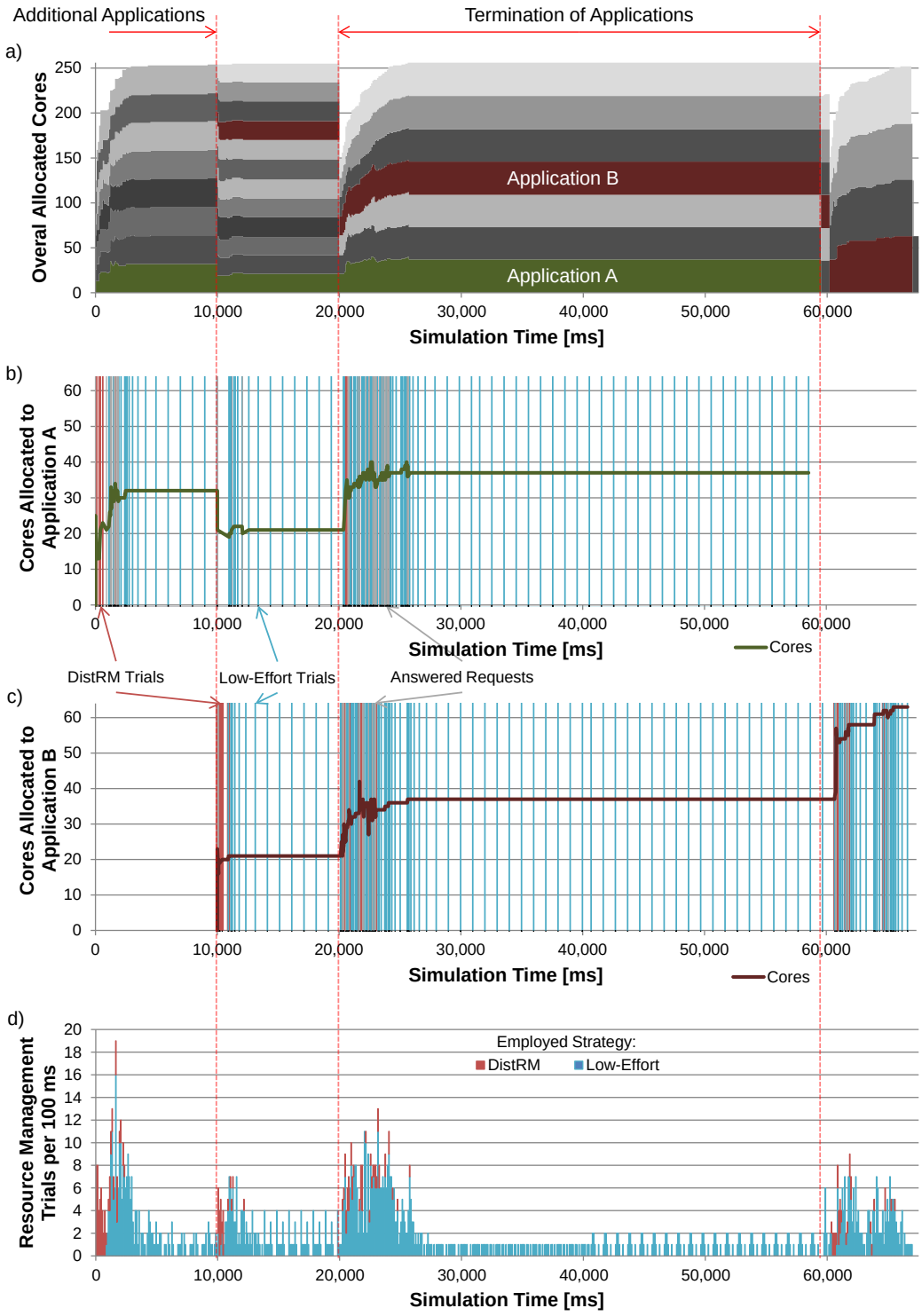

Figure 6.8.: Adaptation to different scenarios of applications that do not change their scalability at runtime. Sub-Figure a) shows the distribution of cores among applications over time, Sub-Figures b) and c) detail application mapping optimization trials and the number of allocated resources for two selected applications A_A and A_B. Sub-Figure d) summarizes the employed resource management trials of all concurrently operating Agents

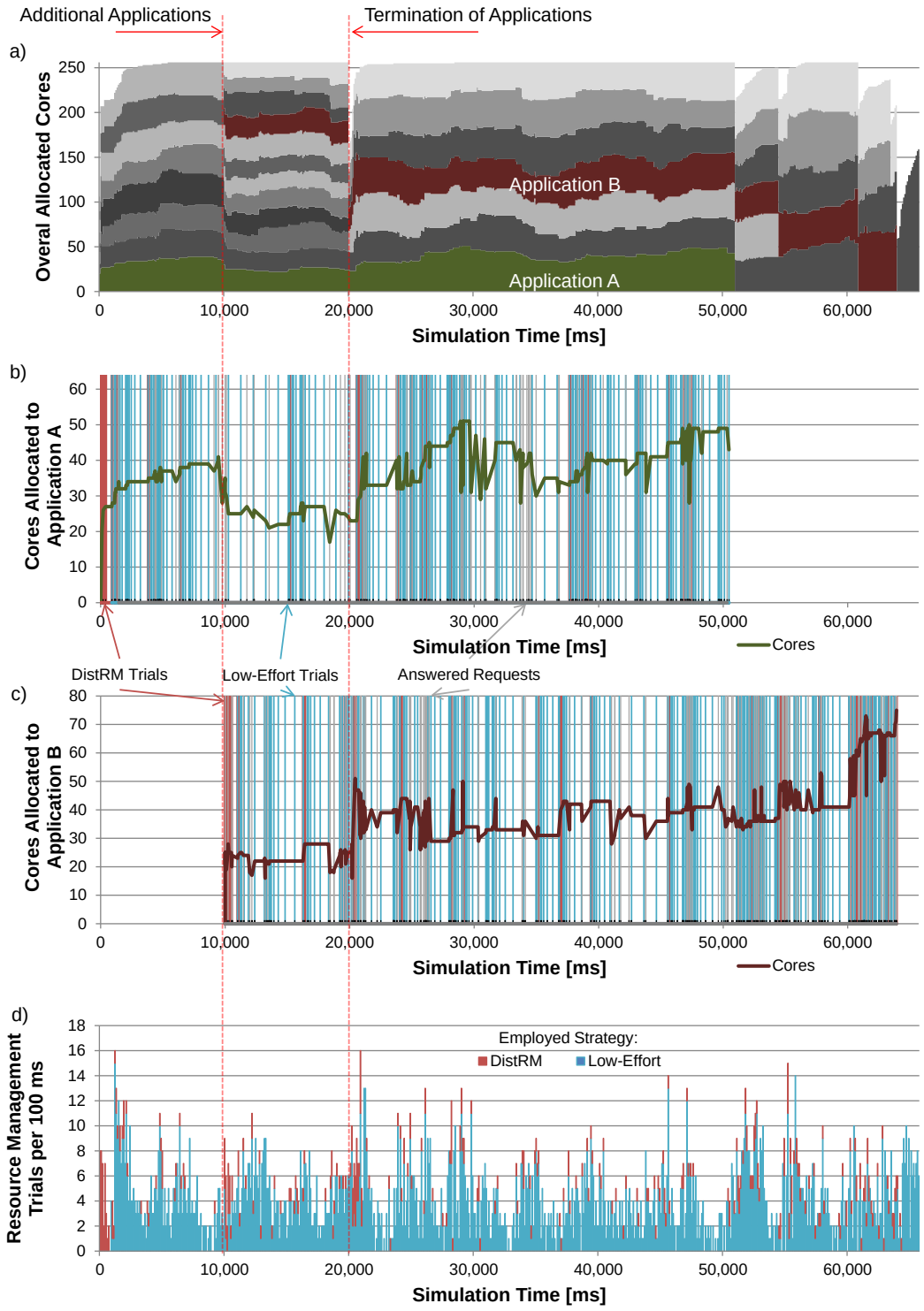

Figure 6.9.: Adaptation to different scenarios of applications that frequently change their scalability at runtime, i.e. no steady application mapping state exists. Sub-Figure a) shows the distribution of cores among applications over time, Sub-Figures b) and c) detail application mapping optimization trials and the number of allocated resources for two selected applications A_A and A_B. Sub-Figure d) summarizes the employed resource management trials of all concurrently operating Agents

low-effort strategy is used for gradual fine-tuning. Once a stable state is reached, the delay between consecutive optimization trials increases. Whenever there is a change in the workload (e.g. at simulation times 10,000ms, 20,000ms, and about 60,000ms), the resource management gets very active to achieve a stable state again. Once this state is reached, the delay between the optimization trials grows again. Sub-Figure d) shows the accumulated resource management trials of all concurrently operating Agents, i.e. the height of the bars shows the sum of resource management trials per 100ms. The employed strategy is indicated by its color code.

A similar analysis is given in Figure 6.9. However in that application scenario, applications consist of multiple phases and frequently change their scalability. This also frequently invokes mapping optimizations as shown in Sub-Figures b), c) and d). As the applications permanently change their scalability, no steady application mapping state exists. Reacting to these frequent changes would not be feasible when using a centralized resource manager that requires several seconds to decide the application mapping. However, frequent changes also lead to an increase of the induced overhead of distributed resource management, as evaluated in the next Section 6.3.3.

The adaptive selection of the employed strategy allows to reduce the overall overhead of distributed resource management without inducing high latencies for e.g. starting the execution of additional applications. Figure 6.10 shows how the reduction of the communication overhead is achieved for an application scenario of 16 concurrent applications that frequently change their scalability, executed on an 256 core system. The figure shows the communication effort required to perform the resource management when using DistRM, AStra, or the low-effort strategy. When using AStra, in the beginning, mostly the DistRM strategy is selected to perform the application mapping (resulting in a communication pattern similar to DistRM). As soon as an initial application mapping has been found, AStra more often selects the low-effort strategy to a) further optimize the application mapping and b) adapt to small changes – eventually leading to a communication pattern similar to the one of the low-effort strategy. As applications complete their computations, the resource management related communication volume gradually decreases. When the last application terminated, the respective line in the chart discontinues.

6.3.3. Scalability Analysis

The results of the scalability analysis of AStra are shown in Figure 6.11. Again, Sub-Figures a), b) and c) show the computation effort for various system sizes from 16 to 1024 cores for 8, 16 and 32 concurrent applications. Sub-Figures d), e) and f) show the respective communication effort, and Sub-Figures g), h) and i) the application mapping quality relative to the centralized resource manager while ignoring its overhead. Different application scenarios are evaluated: In Sub-Figures a), d) and g) the scalability of the applications stays constant over their whole

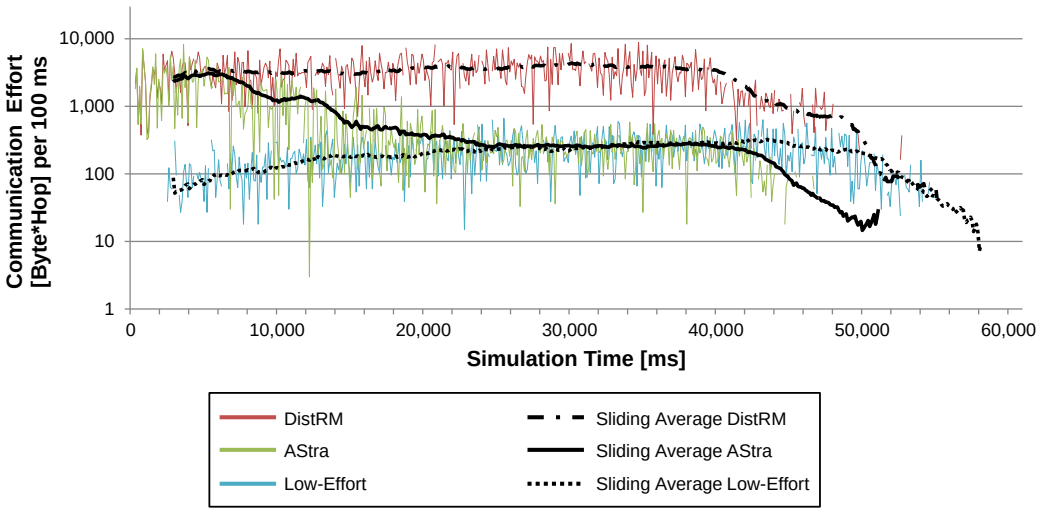

Figure 6.10.: Communication effort required to perform the resource management when using DistRM, AStra, or the low-effort strategy for an application scenario of 16 concurrent applications that frequently change their scalability, executed on an 256 core system

execution time, i.e. they consist of only one phase (see Section 3.2) and a steady state in which no further optimization is meaningful is achievable. In Sub-Figures b), e) and h) the applications change their scalability ten times and in Sub-Figures c), f) and u) 100 times respectively, i.e. they represent (highly) dynamic workload scenarios that benefit from a continuous adaptation of the application mapping.

Similar to the DistRM strategy, the computation and communication effort grows with frequent changes in the workload (Sub-Figures c), f), and i)). However, the effort stays within absolutely feasible bounds compared to centralized resource management. Especially in 1024 core systems the computation complexity of centralized resource management would not allow adopting the application mapping to the frequent changes in highly dynamic workload scenarios in time (see Section 5.3.3 and Section 6.4.1). All application scenarios show that the computation effort as well as the communication effort scale linearly with the number of cores in the system. In case of no (Sub-Figures a) and d)) or moderate (Sub-Figures b) and e)) dynamic in the workload, the effort also scales linearly with the number of concurrently executing applications. In case of highly dynamic workload scenarios (Sub-Figures c) and f)), the effort does not directly correlate with the number of concurrently executing applications, i.e. the resource management is actively optimizing the application mapping all the time anyways, such as additional applications only induce a small additional overhead.

As discussed in Section 6.5, compared to the DistRM strategy, the computation and communication effort is reduced. The application mapping quality is generally very good and close to the application mapping quality achieved by the centralized

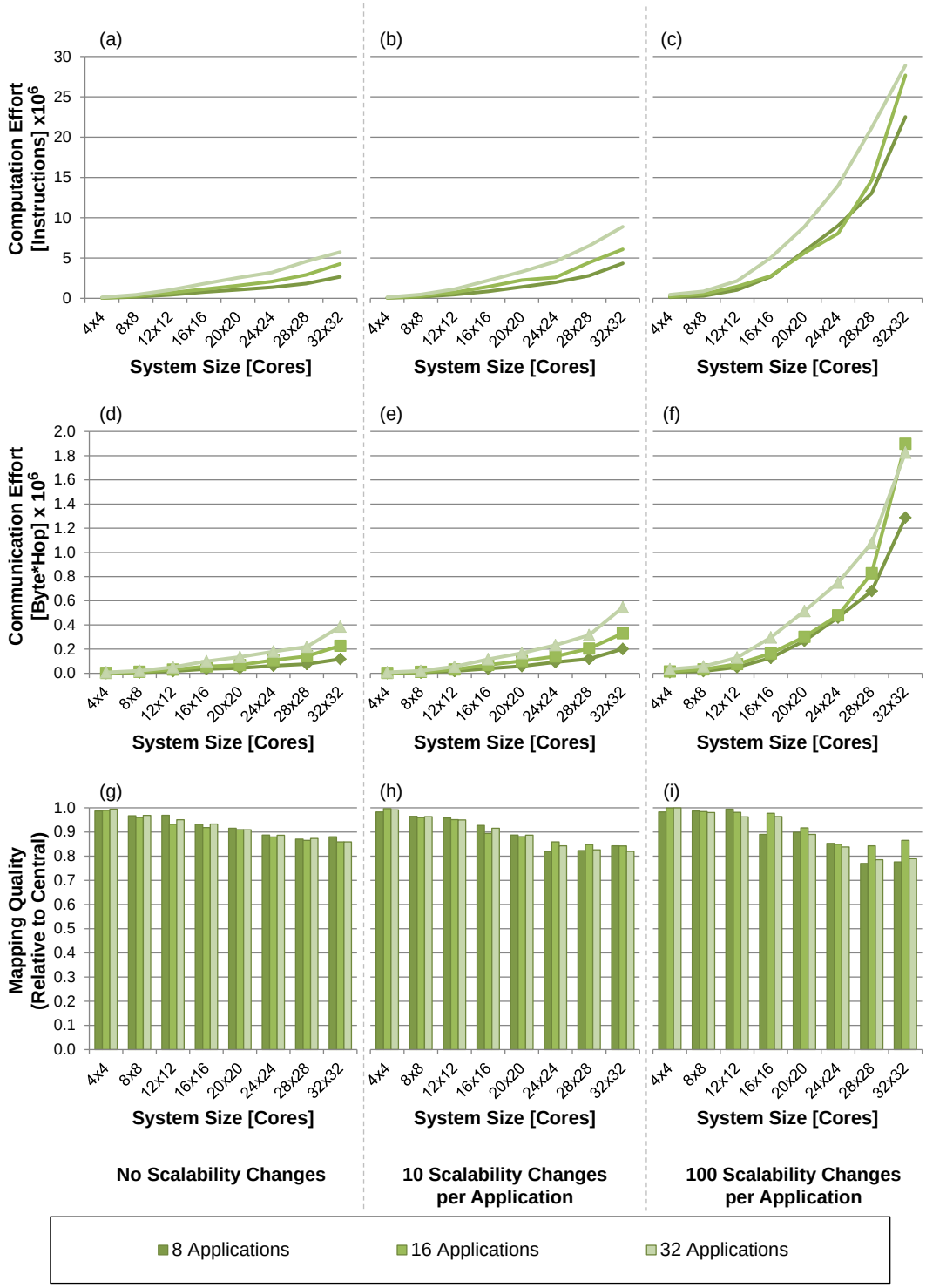

Figure 6.11.: The computation effort (a), b) and c)), communication effort (d), e) and f)) and the relative application mapping quality compared to centralized resource management (g), h) and i)) achieved by AStra for various application scenarios and many-core system configurations

resource management when using a conservative comparison by assuming centralized resource management is feasible and ignoring its overhead and latencies.

6.4. Reference Implementations

The different distributed resource management strategies presented in this thesis are compared to the centralized resource management heuristic presented in Section 5.3. A detailed evaluation of the computation and communication effort of the centralized resource management heuristic is presented in Section 6.4.1. Additionally, the strategies have been compared to the state-of-the-art distributed resource management [ATBS13], presented in Section 5.1.3. A detailed evaluation of [ATBS13] is given in Section 6.4.2.

6.4.1. Centralized Resource Management

The computation effort and communication effort of centralized resource management (as presented in Section 5.3) for the different application scenarios are shown in Figure 6.12. As before, in Sub-Figures a) and d) the scalability of the applications stays constant over their whole execution time, i.e. they consist of only one phase (see Section 3.2) and a steady state in which no further optimization is meaningful is achievable. In Sub-Figures b) and e) the applications change their scalability ten times and in Sub-Figures c) and f) 100 times respectively, i.e. they represent (highly) dynamic workload scenarios.

The centralized resource management is triggered whenever a) a new application is about to start execution, b) an application changes its scalability (i.e. a new phase is about to begin), and c) an application finished its computations. After the centralized resource management computed the updated application mapping, it informs all applications about their updated resource allocation. The frequent communication of the resource manager with all concurrently executing applications causes a high communication effort and might cause a bottleneck situation within the communication infrastructure surrounding the central resource manager.

Additionally, the centralized scheme always aims at the globally best solution independent of the current application mapping (resulting in changes in the mapping of almost every application whenever it is activated), whereas the distributes resource managers presented in this thesis perform local changes such that no complete application has to be migrated to another set of cores. In the presented results free task migrations are assumed, which is not realistic for real world systems – in fact, task migrations are expensive in terms of network bandwidth and migration latency [JAFH11], thus the presented comparison with the centralized scheme is conservative. This also means that a more sophisticated mapping scheme would be used for the

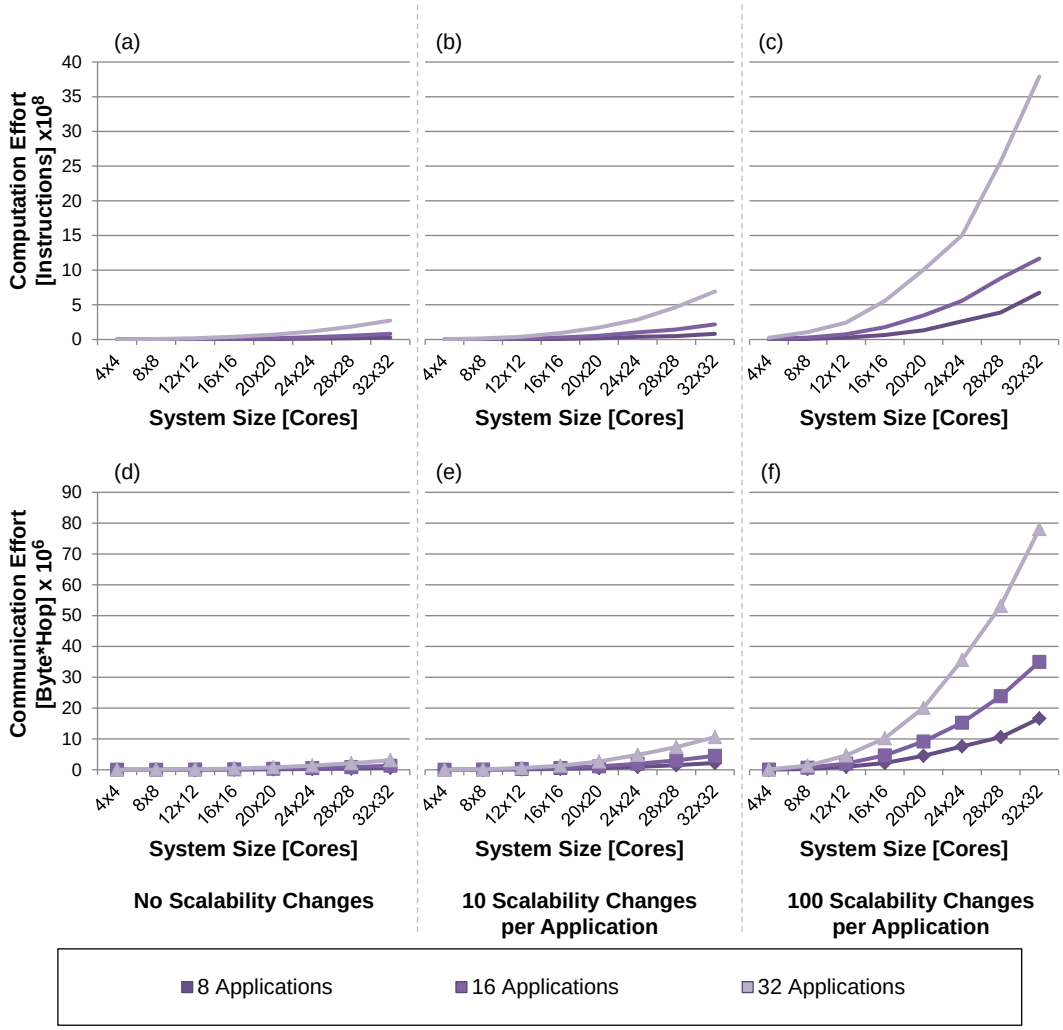

Figure 6.12.: The computation effort (a), b) and c)), and the communication effort (d), e) and f)) required by the centralized resource management for various application scenarios and many-core system configurations

centralized resource manager to reduce the number of task migrations. This would come with a much higher (potentially infeasible) computational effort compared to the used one. In both cases (i.e. frequent task migration or more sophisticated mapping scheme), the application mapping quality of the centralized scheme would be reduced compared to this conservative comparison. This also shows that the slightly decreased application mapping quality achieved by the distributed resource management is acceptable when considering its significantly reduced overhead. As the achieved application mapping quality (without considering the related overhead and latencies) of the centralized resource management is used as reference for all distributed resource managers, the application mapping quality is always 100% and is thus not shown in Figure 6.12.

As shown in Sub-Figures a), b), and c) the computation effort grows quadratically with the number of concurrently executing applications as well as with the number of cores in the system. On average for all analyzed application scenarios and many-core system configurations, the computation effort of the centralized resource management heuristic is more than two orders of magnitude higher than the accumulated effort of all concurrently executing Agents managing the same application scenarios. The computation effort of each individual Agent is less than one thousandth of the centralized resource management. The communication effort (Sub-Figures d), e), and f)) grows about linearly with the number of concurrently executing applications as well as the number of cores in the system.

6.4.2. State-of-the-Art Distributed Resource Management

The protocol used in the state-of-the-art distributed resource management [ATBS13] is quite similar to the DistRM strategy presented in this thesis. However, it uses dedicated cores for resource management that are not available to the applications: one dedicated manager core per application and multiple dedicated controller cores per system. This leads to the situation that e.g. in a system with 64 cores, 4 controller cores, and 16 concurrent applications, only 46 cores are available for the actual applications (see Section 5.1.3. In DistRM and AStra however, the resource management works as a system service interleaved with the applications on the same cores. To allow a fair and conservative comparison, all resource management approaches are implemented with the ability to coexist with applications executing on the same cores, i.e. all cores are available for applications.

Figure 6.13 summarizes the computation effort, the communication effort and the resulting application mapping quality for the different application scenarios. [ATBS13] achieves considerably good application mappings in all application scenarios. To perform the resource management, the many-core system is divided into multiple *regions* of same size. For systems with up to 144 cores, [ATBS13] divides the chip into four regions, for larger many-core systems, the chip is divided into 16 regions. Resource re-allocation optimizations always take place per region and

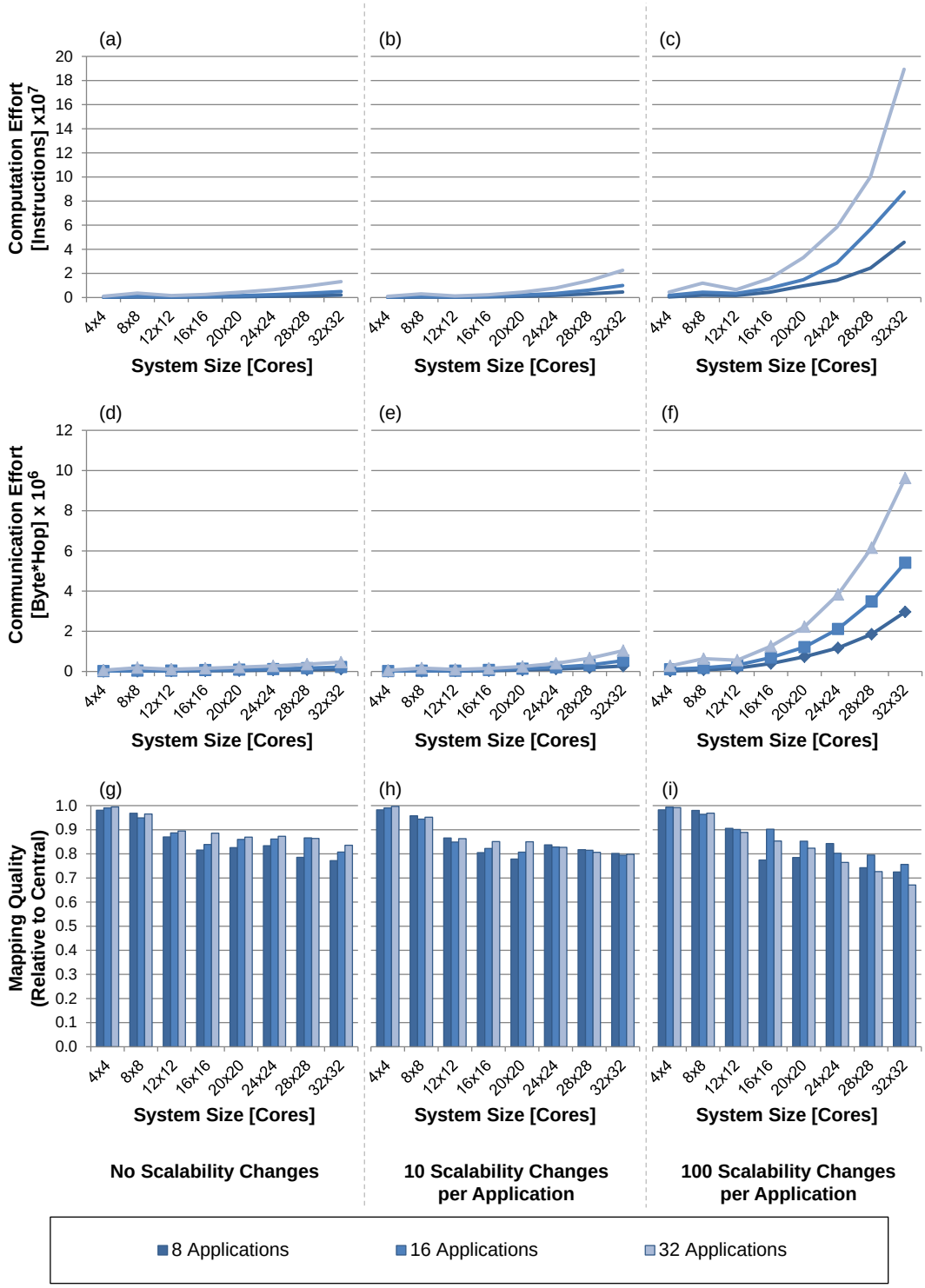

Figure 6.13.: The computation effort (a), b) and c)), communication effort (d), e) and f)) and the relative application mapping quality compared to centralized resource management (g), h) and i)) achieved by [ATBS13] for various application scenarios and many-core system configurations

consider all cores in the region as well as all applications that momentary do have cores allocated within the region. This explains the small peak in computation effort and communication effort in case of the 64 core system.

As the number of cores per region grows linearly with the number of cores in the many-core system and the optimization algorithm has quadratic complexity with respect to the number of cores, the overall computation effort grows quadratically with the number of cores in the system. This limits the scalability of the approach and increases the latency of the individual optimization trials. The communication effort scales linearly with the number of cores in the system (as on average more information is transmitted over longer distances) and with the number of concurrently executing applications. However, the reliance in a fixed infrastructure allowed [ATBS13] to improve the communication effort compared to DistRM.

6.5. Comparison

This Section compares the four different distributed resource management strategies with each other. First, the average values of the achieved mapping quality, the computation effort, and the communication effort for all application scenarios are presented in Section 6.5.1 and Section 6.5.2. A detailed comparison considering the respective application scenarios is given in Section 6.5.3.

6.5.1. Application Mapping Quality

The average application mapping quality of the different distributed resource management strategies is shown in Figure 6.14. As before, the mapping quality is given relative to centralized resource management while ignoring its computational complexity and the resulting latencies evaluated in Section 6.4.1 and Section 5.3.3. An average application mapping quality of e.g. 85% does not look too good at first, however, the computation effort and communication effort to achieve these mappings is feasible in all cases as evaluated in the previous sections and allows to find these mappings with low latencies.

On average for all evaluated application scenarios and many-core system configurations, the low-effort strategy achieves an application mapping quality of 69.5% of the centralized resource management. However, in the worst-case it only achieves 27.6%. Especially on many-core systems with a larger number of cores, the low-effort strategy is not suitable for resource management. When applied to many-core systems with a smaller number of cores it achieves considerably good application mappings (e.g. 89.9% on a 64 core system), however, centralizes resource management is still feasible on theses systems. Table 6.2 summarizes the worst-case, best-case, and the average

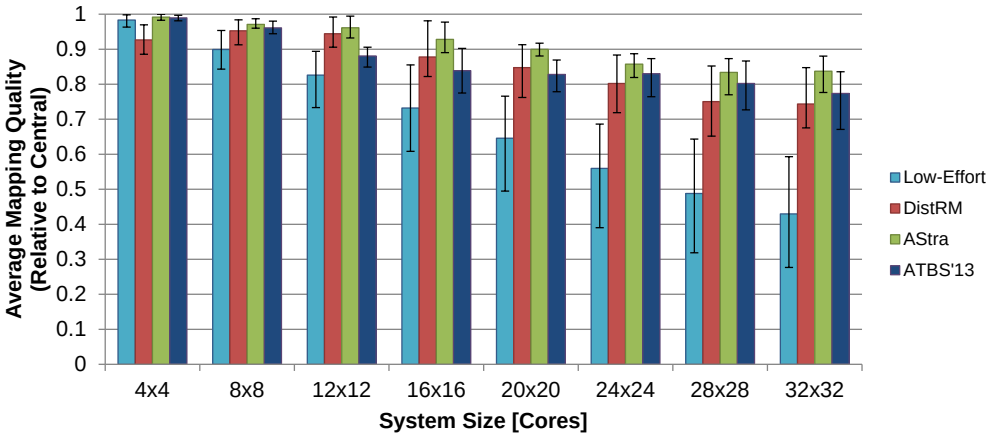

Figure 6.14.: Average application mapping quality of the different distributed resource management strategies in relation to the application mapping quality of centralized resource management, ignoring the feasibility and latencies of centralized resource management

application mapping quality achieved when using the low-effort strategy on various numbers of cores N.

The DistRM strategy achieves application mappings that have on average 85.6% of the application mapping quality which the centralized resource management achieves. Compared with the low-effort strategy, the application mapping quality on average does not degrade significantly with a higher number of cores in the many-core system. However, in the worst-case only 65.2% are achieved. AStra addresses this weakness and improves the worst-case to 77.0%. On average over all evaluated application scenarios and system configurations, AStra achieves 91.0% application mapping quality compared to potentially infeasible centralized resource management. Table 6.3 and Table 6.4 give a summary of the worst-case, best-case, and the average application mapping quality achieved when using DistRM and AStra, respectively.

The state-of-the-art distributed resource management [ATBS13] on average achieves an application mapping quality of 86.3% of centralized resource management. In the worst-case, [ATBS13] achieves 67.1% mapping quality of the centralized resource management. For most application scenarios, [ATBS13] achieves slightly better application mappings than DistRM, however – except on many-core systems with a small number of cores – the application mapping quality is always worse than AStra.

On average AStra achieves a 6.4% better application mapping quality than DistRM and a 5.4% improvement compared to state-of-the-art [ATBS13]. Even more important, the worst-case application mapping quality is improved by 18.1% compared to DistRM and 14.7% compared to [ATBS13]. Additionally, the influence of the application scenario on the mapping quality is reduced from 18.1% (DistRM) respectively 12.3% ([ATBS13]) down to 7.5% when using AStra.

Table 6.2.: Relative Average Application Mapping Quality of the Low-Effort Strategy

N	16	64	144	256	400	576	784	1024
worst-case	96.3%	84.3%	73.3%	60.8%	49.5%	39.0%	31.9%	27.7%
average	98.3%	89.9%	82.6%	73.2%	64.6%	56.0%	48.8%	43.0%
best-case	99.8%	95.3%	89.4%	85.5%	76.6%	68.6%	64.3%	59.3%

Table 6.3.: Relative Average Application Mapping Quality of DistRM

N	16	64	144	256	400	576	784	1024
worst-case	88.6%	91.3%	90.6%	82.2%	76.2%	71.9%	65.2%	67.5%
average	92.7%	95.3%	94.4%	87.8%	84.7%	80.2%	75.0%	74.4%
best-case	97.0%	98.4%	99.2%	98.1%	91.3%	88.3%	85.2%	84.7%

6.5.2. Overhead Comparison

Two categories of overhead are analyzed: a) the computation effort and b) the communication effort.

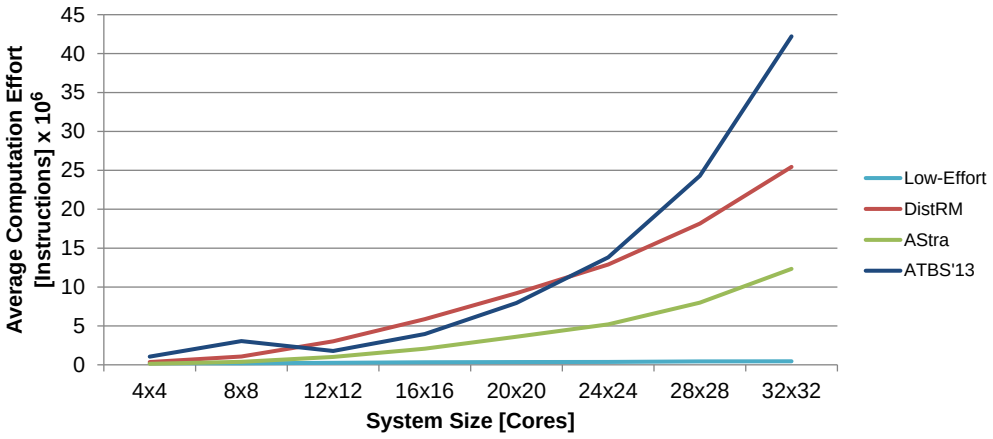

Figure 6.15.: Average Computation Overhead

Figure 6.15 shows the average computation effort of all analyzed application scenarios. On average, AStra is able to reduce the computation overhead by 56.9% compared to DistRM. [ATBS13] achieves the resource management with a lower number of total requests than DistRM. However, these requests also cover larger regions of the chip. As the computational complexity per request scales with the region size, the computation effort is actually higher than the ones of DistRM (on average 29.0%, and up to 65.9% on a 1024 core system) and AStra (on average 199%, and up to 242% on a 1024 core system).

Figure 6.16 shows the communication effort required by the analyzed resource management strategies. Only the resource management related communication is

Table 6.4.: Relative Average Application Mapping Quality of AStra

N	16	64	144	256	400	576	784	1024
worst-case	98.3%	96.0%	93.2%	89.0%	88.1%	81.9%	77.0%	77.6%
average	99.2%	97.1%	96.1%	92.8%	89.9%	85.7%	83.4%	83.7%
best-case	100.0%	98.7%	99.4%	97.7%	91.7%	88.7%	87.3%	88.0%

Table 6.5.: Relative Average Application Mapping Quality of [ATBS13]

N	16	64	144	256	400	576	784	1024
worst case	98.1%	94.4%	84.9%	77.5%	77.9%	76.4%	72.7%	67.1%
average	98.9%	96.1%	88.1%	83.9%	82.8%	83.0%	80.2%	77.3%
best case	99.7%	98.0%	90.6%	90.2%	86.9%	87.3%	86.6%	83.6%

shown, i.e. the application internal communication is omitted. The message sizes s of the low-effort strategy stay constant for all many-core system configurations. Only the average communication distance between interacting Agents grows with the number of cores in the many-core system, resulting in a moderate growth of the communication volume with growing system sizes. DistRM and [ATBS13] have a similar overall communication effort. While [ATBS13] improved the communication scheme and requires less messages to perform the resource management compared with DistRM, the average size of the messages and the average distance between the cores is larger which almost negates the improved communication protocol. AStra reduces the communication effort by adaptively selecting the resource management strategy and often uses the low-effort strategy. This leads, on average, to an reduction of the communication effort by 69.3%. A detailed analysis on how AStra adaptively selects the resource management strategy is given in Section 6.3.2.

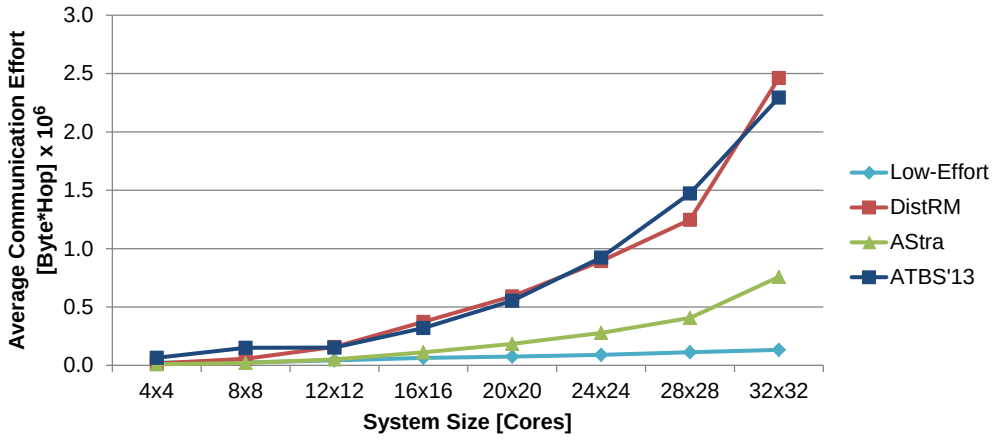

Figure 6.16.: Average Communication Overhead

6.5.3. Detailed Comparison

The properties of the low-effort strategy, DistRM, AStra and [ATBS13] are already evaluated and discussed in Section 6.1, Section 6.2, Section 6.3, and Section 6.4.2, respectively. However, the direct comparison gives a direct visual impression on how the different properties influence the respective effort.

As before, the resource management strategies are evaluated for the various application scenarios and many-core system configurations. Figure 6.17 shows the application mapping quality relative to centralized resource management, Figure 6.18 shows the computation effort, and Figure 6.19 the communication effort. In the respective Sub-Figures a), b), and c) the scalability of the applications stays constant over their whole execution time, i.e. they consist of only one phase (see Section 3.2) and a steady state in which no further optimization is meaningful is achievable. In Sub-Figures d), e) and f) the applications change their scalability ten times and in Sub-Figures g), h) and i) 100 times respectively, i.e. they represent (highly) dynamic workload scenarios that benefit from a continuous adaptation of the application mapping. The number of concurrently executing applications is varied between eight (Sub-Figures a), d), and g)), 16 (Sub-Figures b), e), and h)), and 32 (Sub-Figures c), f), and i)).

6.6. Summary of Evaluation and Comparison

This Chapter gave an comprehensive analysis of the different distributed resource-management strategies presented in this thesis. The linear (or even better than linear) scalability with respect to the number of cores in the many-core system and the number of concurrently executing applications of the adaptive resource-management strategy AStra was shown. Profound analysis of the strategy selection and the adaptive optimization delay and the influence of the different configuration parameters completed this analysis. On average, AStra achieves an 5.4% improved application mapping quality compared to state-of-the-art distributed resource management [ATBS13] and requires only 33.4% of the computation effort and 30.7% of the communication effort for the same application scenarios and many-core system configurations. The computation effort of the centralized resource management heuristic presented in Section 5.3 is more than two orders of magnitude higher than the accumulated computation effort of AStra handling the same application scenarios. On average, AStra still manages to achieve 91.0% of the application mapping quality when using a conservative comparison and ignoring the overhead and latencies of the centralized resource manager.

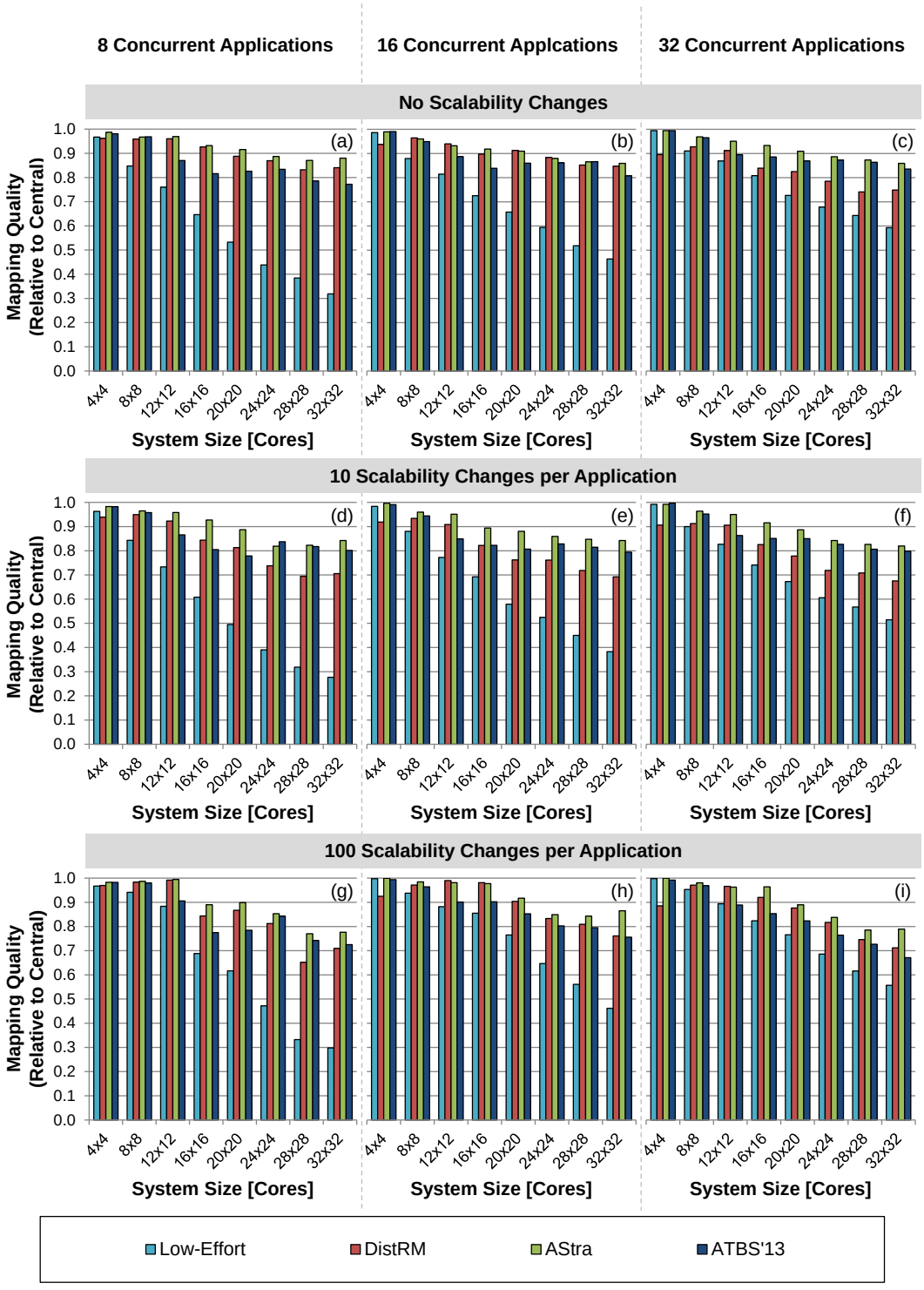

Figure 6.17.: Comparison of the achieved relative application mapping quality of the different distributed resource management strategies for the various application scenarios and many-core system configurations

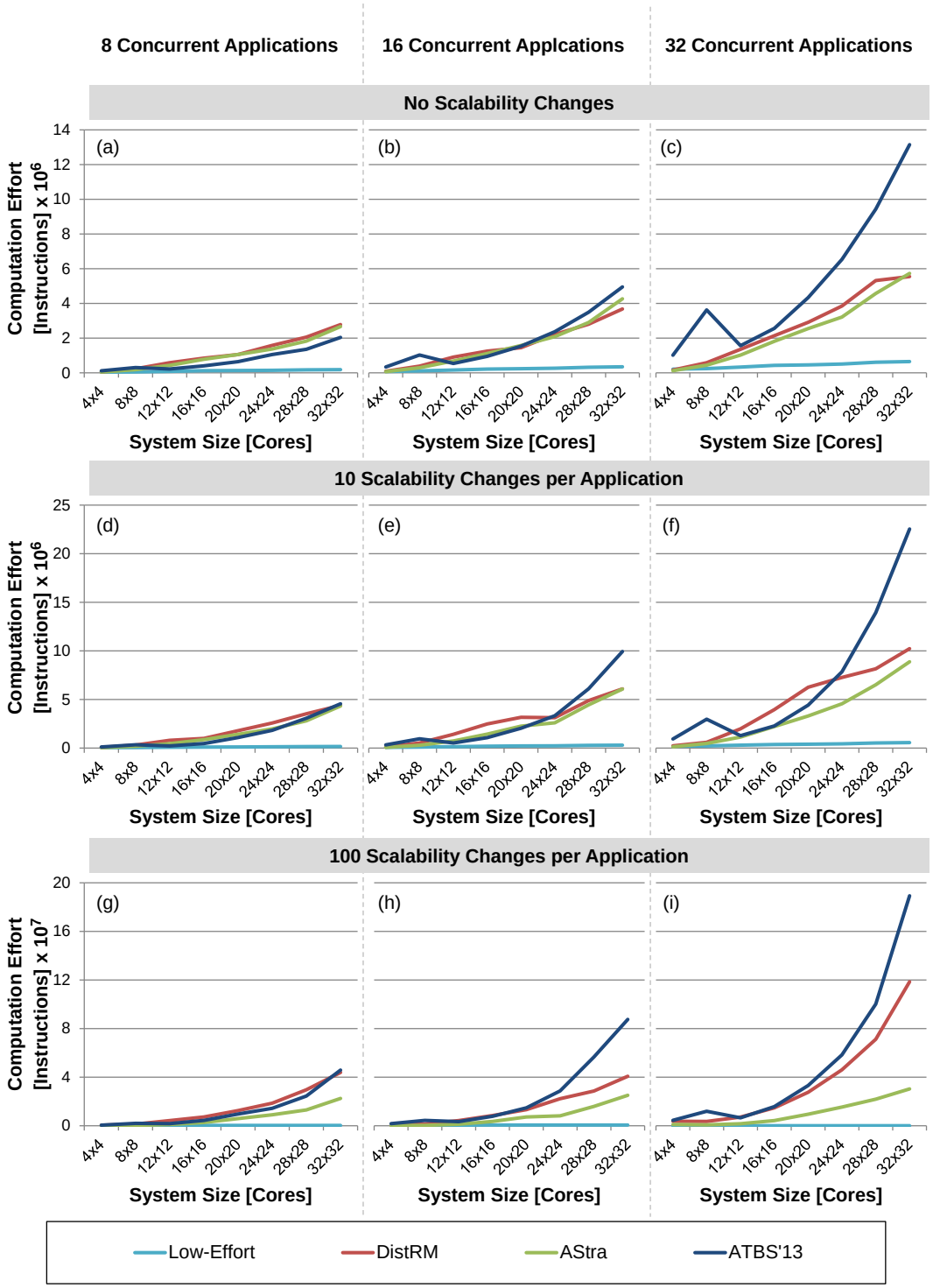

Figure 6.18.: Comparison of the computation effort of the different distributed resource management strategies for the various application scenarios and many-core system configurations

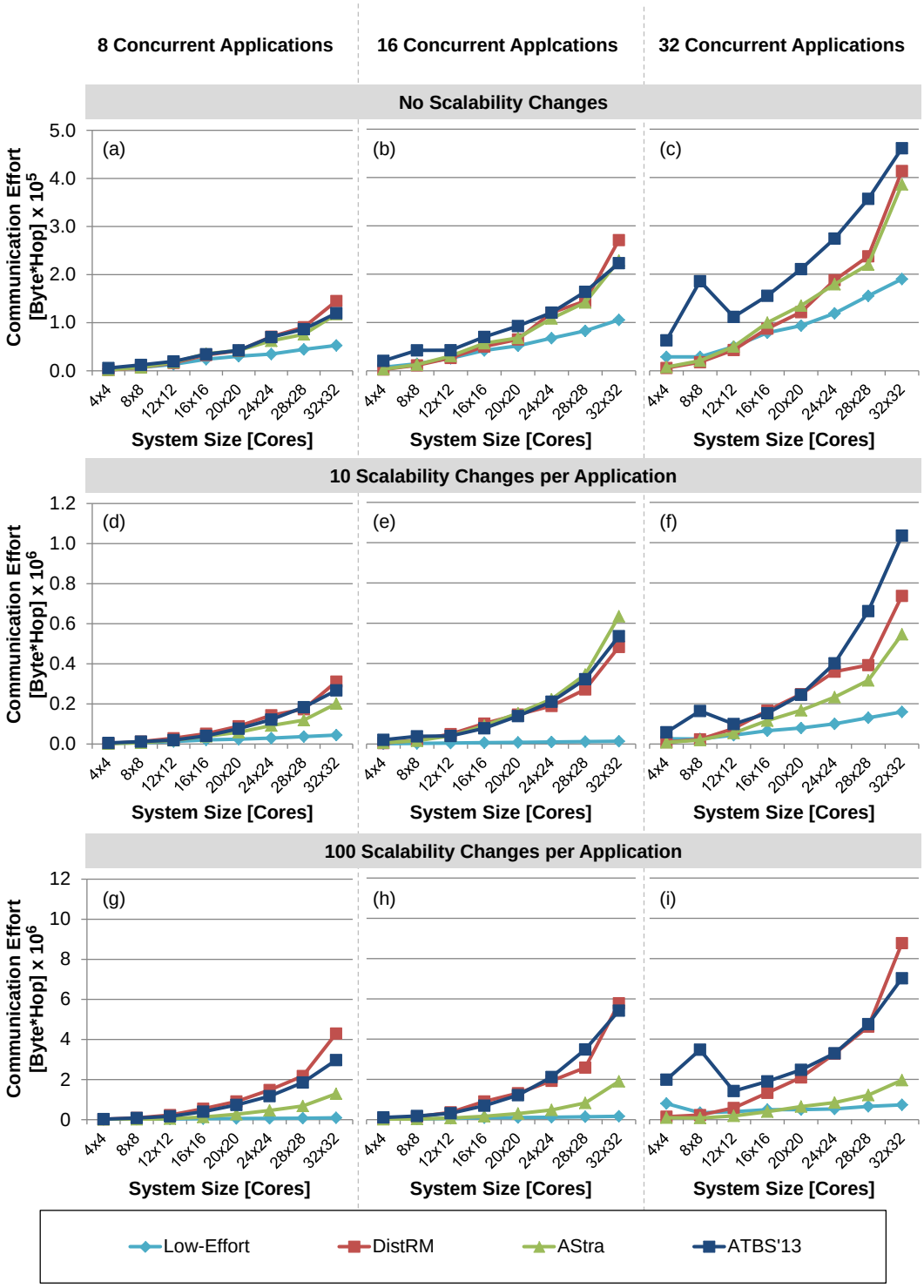

Figure 6.19.: Comparison of the communication effort of the different distributed resource management strategies for the various application scenarios and many-core system configurations

7. Conclusion and Outlook

To conclude this thesis, this chapter first recapitulates the contributions of this thesis. Then, potential future research topics based on top of the scalable distributed resource management are presented and discussed.

7.1. Thesis Summary

Managing the resources of a large on-chip many-core system (a NP-Hard problem) centrally leads to serious issues in terms of computational complexity and thus high latencies when the number of cores gets too large. The computational complexity is caused by evaluating different potential application mappings based on an application performance model as well as the decision making process itself. In order to address these issues, an adaptive on-the-fly application performance model and a fully distributed resource management were developed in the scope of this thesis. This thesis has shown that the presented distributed resource management for many-core systems is a suitable means to address the scalability issues of centralized resource management with an increasing number of cores and concurrently executing applications.

When deciding an application mapping, not only the number of cores allocated to each application is of importance. Because of communication latencies between the individual application tasks mapped to these cores also the selection of these cores and their relative topological position on the chip matters. The adaptive on-the-fly application performance model considers the topological properties of the cores allocated to an application in NoC based many-core systems. It uses a simple metric that can easily be determined to estimate the application performance based on the lower bound and the upper-bound performance of the application. To handle highly dynamic behavior of workloads not known a priori, the application performance model is continuously adapted at runtime. The improved accuracy of the performance estimations results in overall high execution efficiency when the model is used for application mapping decisions. The evaluations show that the average estimation error is reduced from 14.7% to merely 4.5% while at the same time significantly improving the accuracy and reducing the worst-case error. As a result, the applications profit from better mappings compared to state-of-the-art. The on-the-fly application performance estimation model enables managing many-core

systems that exhibit rapid and spontaneous workload variations while maintaining the mapping quality.

To make the decision making for resource management highly scalable, the principles of a Multi Agent System [Wei99] are employed and one dedicated Agent per application is used as resource manager. The computations of an Agent are performed on the same cores as the application and thus, the overall computational effort of the resource management is distributed throughout the many-core system. This provides important advantages such as inherent parallelism and the avoidance of computational bottlenecks as only the local application mapping is considered and optimized. Additionally, the communication of the resource management Agents occurs locally in different regions of the chip instead of concentrating in one point. The combined advantages of the on-the-fly application performance modeling and the Multi Agent System make the resource management scalable and less intrusive to the actual applications.

The initial resource management strategy presented in this thesis **DistRM** [KBL+11] was designed as a distributed system without any global synchronization or global communication. Therefore it scales with the size of the many-core system and the number of applications. Results show that DistRM works as well in 64 (8x8) core systems as in 1024 (32x32) core systems. The design principles of DistRM inspired other research groups that adapted the ideas and improved upon them, e.g. [ATBS13].

DistRM is based on periodic application mapping optimizations that trigger the analysis and modification of multiple application mappings to achieve coarse changes with a low latency. However, the overhead that is required to perform small changes in the application mapping is as high as the overhead for coarse changes, e.g. to determine the initial application mapping. This overhead leads to a large delay between two optimization runs once an initial application mapping has been found – DistRM is therefore not optimized to react to frequent small changes in the application resource demands.

To address this weakness, this thesis combines the *complex DistRM strategy* with another *low-effort strategy* in order to implement an *adaptive strategy* (**AStra**) for low overhead distributed resource management. The main idea behind it is as follows: to compensate the weaknesses and to amplify the strengths of these two resource management strategies, AStra adaptively switches between these two basic resource management strategies at runtime.

As a result, AStra was the first approach to employ two inter-operable strategies for distributed resource management to achieve a highly efficient resource management. To select the employed strategy, it exploits application knowledge on a) the best-suited number of cores $N_{A_i}^{\Phi_{max}}$ for an application A_i at any time during execution, b) the local system state, and c) the set of cores C_{A_i} already allocated to application A_i.

The proposed adaptive strategy AStra is able to flexibly react to changes of the

resource demands of an application and to find good application mappings fast. It can optimize the application mapping gradually with a very low overhead. Compared to state-of-the-art distributed resource management [ATBS13], the overall application mapping quality is improved by 5%. At the same time, the computational effort is reduced by 67% on average and the communication effort by 69% on average. The computation effort of a centralized resource management heuristic is more than two orders of magnitude higher than the accumulated computation effort of AStra handling the same application scenarios. On average, AStra still manages to achieve 91.0% of the application mapping quality when using a conservative comparison and ignoring the overhead and latencies of the centralized resource manager. This makes AStra an optimal choice for managing the resources of future many-core systems with low latencies and a low overhead.

7.2. Future Work

The fully distributed architecture of the resource management presented in this thesis not only enables highly scalable resource management, it also allows improving the reliability of the system. As there is no single point of failure, parts of the chip may fail without having an impact on the still operational parts. The protocols used for resource management only have to be extended slightly to deal with lost or corrupted messages. Obviously, applications executing on a failed piece of hardware will produce no results, or – even worse – wrong results. Therefore, a goal of future work might be to detect failing hardware before they actually fail and not use them for application mapping. This might entail building a dynamic 'chip map' that contains information on which resources are available for application mapping. When coupled with a dynamic resource discovery, this also allows a plug-and-play style addition of hardware components to the Network-on-Chip.

Due to the power density of current and future technology nodes and the resulting thermal issues, future many-core systems are not expected to be able to operate all of their cores at the same time – a problem known as 'dark silicon'. Dark silicon management includes deciding which resources to operate a which performance levels and will become an integral part of resource management. To enable resource management avoiding (temporally) unavailable cores due to dark silicon management decisions, a similar concept to the previously mentioned dynamic chip map might be used. A more scalable solution might incorporate specialized Agents that represent the dark silicon management decisions. While they might implement the regular protocols used by the Agent System, they could allocate individual cores that are affected by dark silicon management on purpose and will not give them away until the system state allows their utilization for application execution again.

Another research topic is the extension of the application performance estimation towards heterogeneous computation resources and communication channels to the

memory. Based on the application implementation and the kind of heterogeneous resources, the application might only support execution on certain kinds of resources or perform suboptimal on the 'wrong' kind of hardware. The challenge here lies in finding a parameterizable generic representation of the application and the hardware platform that is not specific to an individual application. Also, the performance estimation should still be lightweight in order to avoid high decision latencies.

Finally, a research direction might be the translation of quality requirements requested by the application (such as the desired degree of predictability, the level of timeliness or the amount of performance for a certain computation period) to actual resource demands and the resulting application mapping. This would allow application designers to specify typical application demands (e.g. the execution of a specific part of the application 25 times per second) without knowing the hardware properties of the many-core system and use these demands for application mapping optimization instead of 'blindly' improving the overall application performance, e.g. with respect to dark silicon optimizations.

Appendix

A. Many-Core and Multi-Agent Simulation

Two simulation environments were used to evaluate the resource management presented in this thesis. The first one is a system-level simulation that allows for relatively fast design space exploration. It is presented in Appendix A.1. All presented protocols and algorithms are implemented and developed in this simulator. It uses source-code annotations to estimate the computation latencies and overhead. The estimations are based on an implementation in a cycle-accurate many-core simulator (Appendix A.2) and an implementation on the Intel SCC (see Section 2.1.2, [PKH12]). All results presented in Chapter 6 are obtained by system-level simulation.

A.1. System-Level Simulation

A simplified class diagram of the system-level many-core simulator is shown in Figure A.1. Especially the GUI (Graphical User Interface) is omitted to avoid cluttering of the diagram. The GUI is very useful for analyzing the application mapping decisions and developing resource management protocols. A screenshot of such a simulation is shown in Figure A.2. To accelerate simulation, the GUI may be completely disabled.

The core of the simulator is the central `Event Queue` which manages the momentary simulation time and holds a queue of `Event`s that await processing. Each `Event` has a timestamp which indicates at which simulated time it should be handled. Events may be scheduled for later points in simulation time or may be enqueued to the momentary simulation time. Once all events that had been scheduled to the momentary simulation time are handled, the simulation continues with the earliest next enqueued `Event`. Most components in the simulator interact through events. Therefore, they inherit specialized classes from the generic `Event` class that indicates the desired payload and the corresponding `Event Handler`. Therefore, most components of the simulator inherit from the `Event Handler` class.

As an example for using the event queue, assume an Agent wants to send a `REQUEST` message through the NoC to an other Agent. Therefore it creates a `AgentRequestMessage` object that contains the necessary information to perform the request. The `AgentRequestMessage` class inherits from `Message` class that

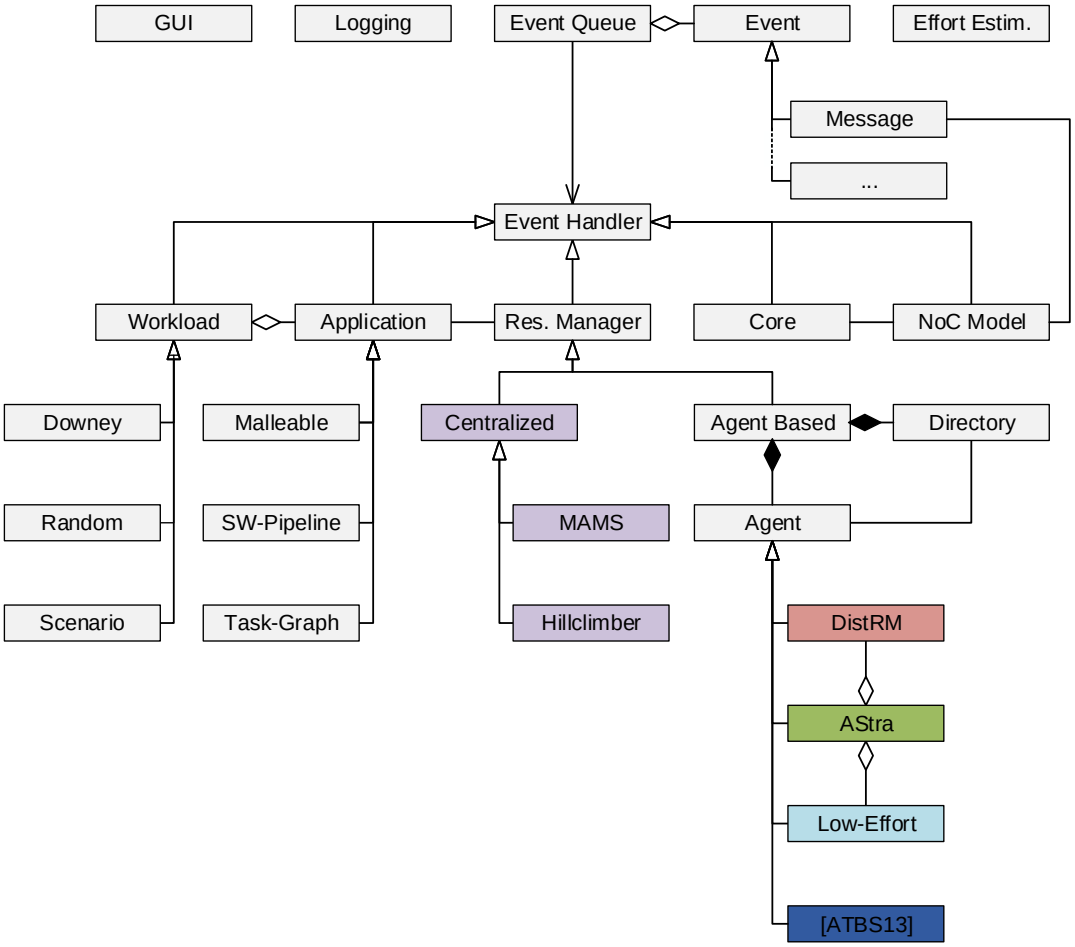

Figure A.1.: Simplified class diagram of the system-level many-core simulator used in this thesis

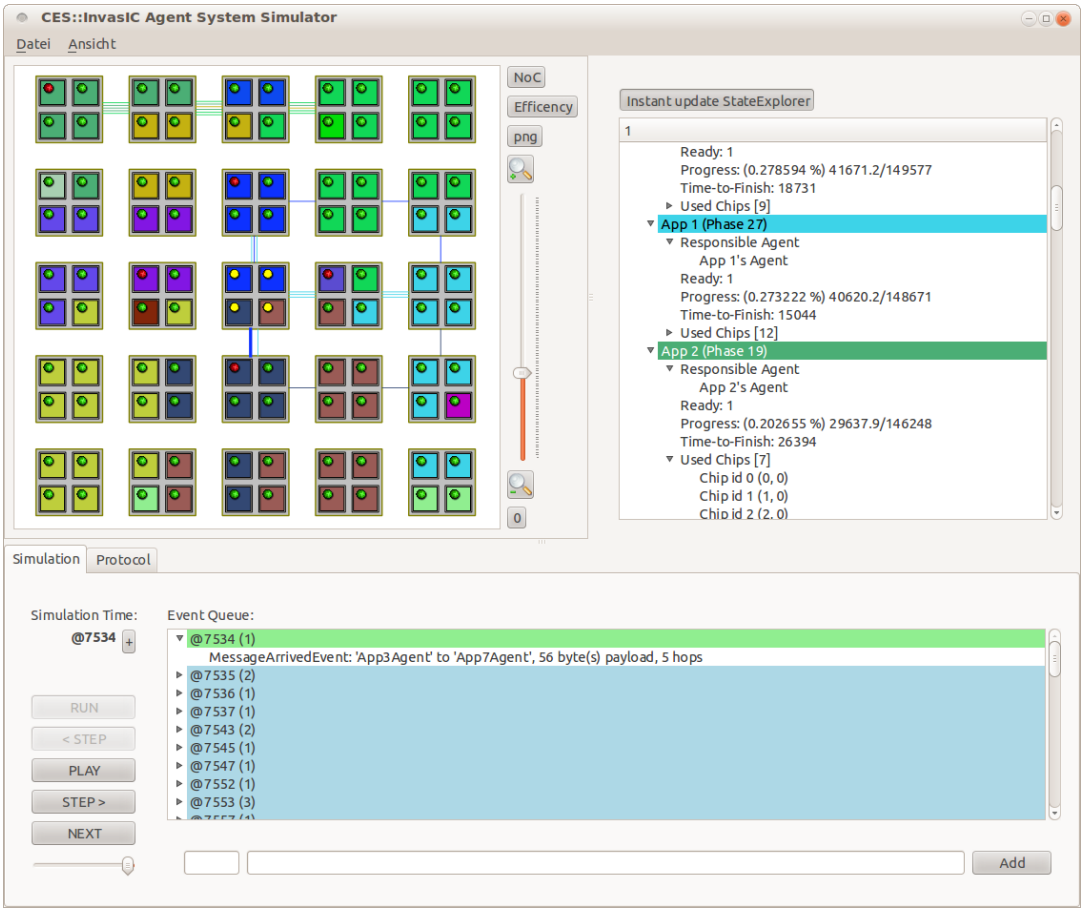

Figure A.2.: Screenshot of the system-level simulator

contains the information required by the NoC. The `Message` class itself inherits from the `Event` class which allows to use the Event Queue. The `AgentRequestMessage` message is given to the NoC model which simulates the path of the message through the NoC. When the message arrived at the targeted core, the specialized `Event Handler` of the addressed Agent is executed.

In the following Subsections, the most important components of the Simulator are presented.

A.1.1. Workload Models

The workload model is responsible for instantiating new applications and for purposefully terminating applications to dynamically generate stress for the resource management. Different workload generators have been implemented:

- A parser for workload configuration files produced by Downey's workload generator (see Appendix B.1). The generated workload resembles the workload generally found in high performance computing environments and is widely used (e.g. [ZWB00, CB00, Fei03, SCH11, SLS07, PYL13, ATBS13]) for evaluating many-core resource management.
- A workload generator that produces random workloads that emulate interactive utilization of a many-core system
- A scenario based workload generator that produces predefined workload patterns, e.g. an always growing/shrinking number of applications, or repetitive impulses, etc.. These scenarios allow to evaluate corner-cases in resource management.

A.1.2. Application Models

The application model is responsible for generating computation load on the cores, for generating traffic in the NoC, and of course for triggering the resource manager whenever the applications resource demands change. Depending on the frequency of these changes, the stress for the resource manager grows with a growing number of concurrent applications. The application model provides a generic interface to the resource manager, independent on the actual implementation of the application model. The following kinds of application models are implemented:

- A model for malleable applications. The model is based on Downey's application model [Dow98], however, it uses the application performance model presented in this thesis to calculate the speedup instead. To increase the stress on the resource manager, the applications are separated in different phases with different scalability. The resource manager is triggered whenever a new phase

is about to begin to allow an optimization of the application mapping. The model allows adding and removing cores to/from the set of cores allocated to an application at any point in time.

- The software-pipeline model presented in Section 2.2.4. It uses application traces of real applications for parametrization (see Appendix B.2). The model allows adding cores to the set of cores allocated to an application at any point in time. Cores may be removed from the set of cores allocated to an application whenever the pipeline stage currently assigned to these cores completed the execution of the momentary iteration. Communication between the pipeline stages use the NoC model.
- A task-graph application model [KKH13]. It includes a parser for task-graph definitions in the TGFF format (see Appendix B.3). It uses the scheduling heuristic presented in [THW02] to schedule the execution of tasks on assigned cores. Communication between the application tasks use the NoC model.

A.1.3. Resource Management Models

The most important component in the simulator is the resource management model. It receives requests from applications and uses on of the implemented resource managers to decide the application mapping based on these requests. Various resource managers have been implemented and evaluated in the simulator. The most important ones are presented in detail in this thesis:

- Centralized resource managers decide the mapping of all applications at once. Two centralized resource managers have been implemented:
  - The hill-climbing heuristic, presented in Section 5.3.1
  - State-of-the-art centralized MAMS multi application multi step mapping method, presented in Section 5.3.2.
- Distributed Agent based resource management and the necessary infrastructure (see Section 5.4.1). Distributed resource managers decide the mapping of local areas on the chip for only a few applications. Multiple decisions take place concurrently and repeatedly. The Agent based resource management uses different strategies to achieve the resource management. Implemented strategies are:
  - The DistRM, presented in Section 5.4.3
  - The low-effort Strategy, presented in Section 5.4.5
  - The adaptive Strategy AStra, presented in Section 5.4.6
  - State-of-the-art distributed resource management [ATBS13], presented in Section 5.1.3

A.1.4. NoC Model

The NoC model simulates communication in the many-core system. It uses the event queue to propagate messages through the NoC. Once a message has received the target core, an event is enqueued to trigger an action in the receiving component. The NoC model uses xy-routing. Measurements using the Network-on-Chip of the Intel SCC (see Section 2.1.2) were performed to improve the delay calculation [PKH12].

A.2. Cycle-Accurate Simulation

Additionally to the system-level simulation, parts of the Agent System have been implemented [BKH14] within the cycle-accurate many-core simulator Hornet [LRC+11]. Hornet is a NoC simulator that includes a MIPS CPU simulator per core in the NoC. This allows to execute native MIPS code (e.g. compiled C code) in a cycle-accurate bare-metal environment. Hornet provides a small library that allows receiving and sending messages through the NoC. Thus, the Agent System was re-implemented to run in this environment. The resulting software architecture is shown in Figure A.3.

The Hornet simulator was enhanced [BKH14] to allow the measurement of the execution time of individual parts of the executed application by means of syscalls. This allowed to obtain cycle-accurate evaluations of the algorithms used by the Agents in relation to various input data and parameter configuration. These measurements augment the effort estimation performed in the system-level simulator.

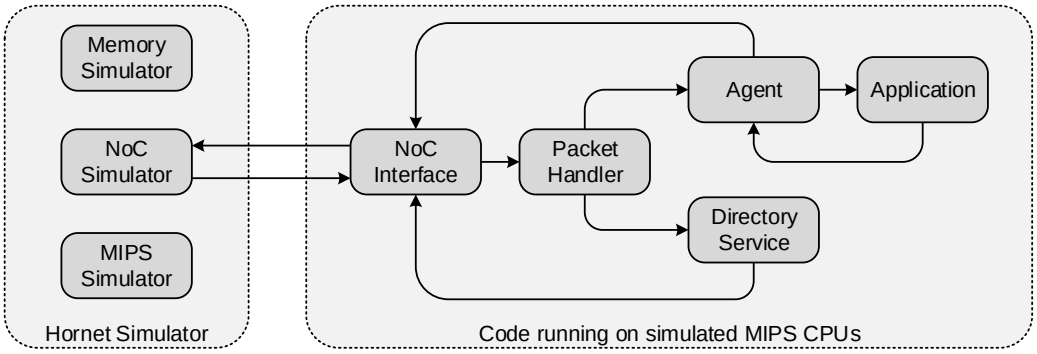

Figure A.3.: Software architecture of the cycle-accurate simulation of the Agent System

B. Workload Parameters

To evaluate the resource management and the performance estimation model presented in this thesis, synthetic workload as well as application traces was used. In the following, Downey's workload generator, malleable software pipelines and the Task Graphs For Free [DRW98] utility are briefly presented.

B.1. Downey's Application Model

A generic application speedup model for parallel workloads has been presented in [Dow98]. It models achievable speedup of real-world applications when executed on computer clusters very well. The model has been validated against various benchmark applications on shared- and distributed memory systems. The parameters of the model correspond to measurable program characteristics. This insight into the values and distributions of these parameters in a real workload allowed to implement a workload generator [Fei05] which allows to generate application profiles and workload scenarios similar to those found in parallel high-performance computing clusters. The generated workload resembles the workload generally found in high performance computing environments and is widely used (e.g. [ZWB00, CB00, Fei03, SCH11, SLS07, PYL13, ATBS13]) for evaluating many-core resource management. To evaluate the distributed resource management presented in this thesis, workload profiles generated by [Fei05] were used. Therefore, to generate suitable application profiles, the workload generator had to be compiled for different system sizes, resulting in different binaries `Workload_4`, `Workload_8`, ..., `Workload_32`. These binaries were executed multiple times to generate various workload profiles. In the following, the script used to run execute these binaries as well as one example of a generated workload profile are shown.

File: generate_workload.pl

```
#!/usr/bin/perl

my @Cores = (4*4, 8*8, 12*12, 16*16, 20*20, 24*24, 28*28, 32*32);
my @Procs = (8, 16, 32);

for ($run=0; $run<10; $run++) {
    foreach $NumCores (@Cores) {
        foreach $NumProcs (@Procs) {
            `./Workload_$NumProcs $run 0.75 0 10 $NumCores >
                files/WL_$NumCores\_$NumProcs\_$run`;
        }
    }
}
```

File: WL_16_32_0 (Generated Workload Profile)

ReadyTime	Total Workload	Parameter P	Parameter s
0.000	4455.42	8.91	1.57
1.061	7.21	6.42	1.54
7.837	118.39	32.69	0.73
8.398	9527.62	33.96	1.43
14.126	1.18	3.85	0.27
35.389	55.12	2.05	0.22
52.850	168.90	104.90	1.23
58.013	189.21	15.44	1.95
60.991	193.92	71.47	0.80
75.459	33.94	88.14	1.84
76.056	192.47	1.61	0.38
77.390	32.75	1.43	0.04
109.087	10.84	217.53	1.80
124.250	220.88	8.01	1.52
128.883	203.58	1.24	0.88
129.705	1352.25	4.84	1.48
141.616	971.28	2.51	0.88
143.765	27.20	3.56	1.79
148.077	14252.43	26.16	1.31
157.507	10297.93	9.11	1.63
158.576	124.57	3.31	1.90
180.520	6705.09	34.98	0.86
195.079	2591.57	5.50	0.89

```
214.280          15.84          21.88          0.83
215.403           2.81           2.01          0.99
215.579       11499.28          44.49          0.77
227.026          18.95           3.63          1.17
248.608        1512.45           2.01          1.59
251.984           2.11         194.12          0.11
271.899          11.03          83.42          1.47
272.279         598.59          67.55          0.19
279.776           2.19           1.47          0.41
```

B.2. Malleable Software Pipelines

Software pipelines (see Section 2.2.4) are a well-established means to parallelize stream-processing applications, among which are very common image/video and networking applications [TKA02]. They consist of multiple stages, each processing subsequent iterations on a stream of input data. Each stage i requires the time c_i to compute each iteration. The output data o_i of one stage i forms the input data e_{i+1} of its direct successor. There is no further communication. A malleable software pipeline [JPK+13] can reduce the number of its stages (and thus the number of cores used) at runtime by fusing consecutive stages (see Figure 2.6) so they can be mapped to the same core. The "robotic vision" application is such a malleable software pipeline. The following (shortened) trace (from [JPK+13]) has been generated on the Intel SCC many-core processor ([HDH+10], see Section 2.1.2).

File: RoboticVisionPipeline.txt (Application Trace)

```
# PIPELINE TRACE
NUM_STAGES = 20
NUM_ITERATIONS = 900
STAGE = 0
AVG_HEAP_SIZE = 22729595
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 0
AVG_COMMVOL = 4
STAGE = 1
AVG_HEAP_SIZE = 22729595
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 15740
AVG_COMMVOL = 10
STAGE = 2
AVG_HEAP_SIZE = 22729595
```

```
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 15513
AVG_COMMVOL = 14
STAGE = 3
AVG_HEAP_SIZE = 22729581
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 2533
AVG_COMMVOL = 14

...


STAGE = 16
AVG_HEAP_SIZE = 23656243
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 216
AVG_COMMVOL = 926648
STAGE = 17
AVG_HEAP_SIZE = 23656243
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 216
AVG_COMMVOL = 926648
STAGE = 18
AVG_HEAP_SIZE = 23656243
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 198
AVG_COMMVOL = 1838126
STAGE = 19
AVG_HEAP_SIZE = 20891469
AVG_STACK_SIZE = 40960
AVG_RUNTIME = 1560
AVG_COMMVOL = 0
```

B.3. TGFF: Task Graphs For Free

The TGFF [DRW98] utility generates task graphs, e.g. to evaluate scheduling or resource management algorithms. In this thesis, three different task graphs were generated to evaluate the resource-aware application speedup model presented in Section 4.6.2.

The total workload of the three applications and the number of tasks in each

application is almost the same; they only differ in their communication density. The first task-graph represents an application with sparse communication. Each node in the task-graph (except the end node per phase) only depends on a few predecessors, which allows the task mapper to map the tasks in a way that avoids frequent communications across different cores. The second application features a medium communication density. There are up to five inputs to each node required before the node can execute. This also means that the topological location of the allocated cores has a bigger influence on the speedup. The third task-graph represents an application with very dense communication between the individual tasks. For this task-graph, the relative topological location of the allocated cores is of upmost importance. In the following sections, the TGFF configuration files as well as the resulting task graphs for these three applications are shown.

B.3.1. Sparse Communication

File: sparse_communication.tgffopt

```
tg_cnt 1
task_cnt 300 1
task_degree 1 30
tg_write
eps_write

table_label COMMUN
table_cnt 1
table_attrib price 80 20
type_attrib exec\_time 50 20
trans_write

table_label EXEC
table_cnt 1
pe_write
```

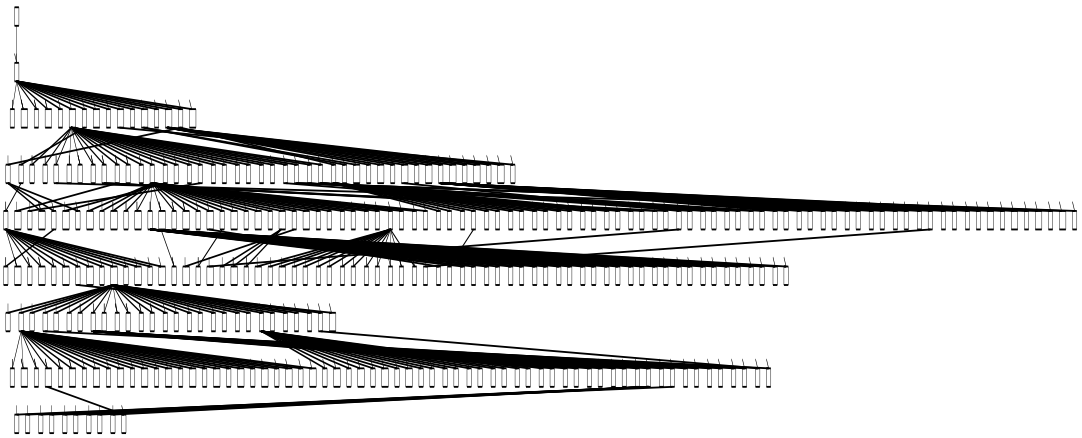

Figure B.1.: Task Graph generated by TGFF with a sparse communication pattern.

B.3.2. Medium Communication

File: medium_communication.tgffopt

```
tg_cnt 1
task_cnt 300 1
task_degree 5 30
tg_write
eps_write

table_label COMMUN
table_cnt 1
table_attrib price 80 20
type_attrib exec_time 50 20
trans_write

table_label EXEC
table_cnt 1
pe_write
```

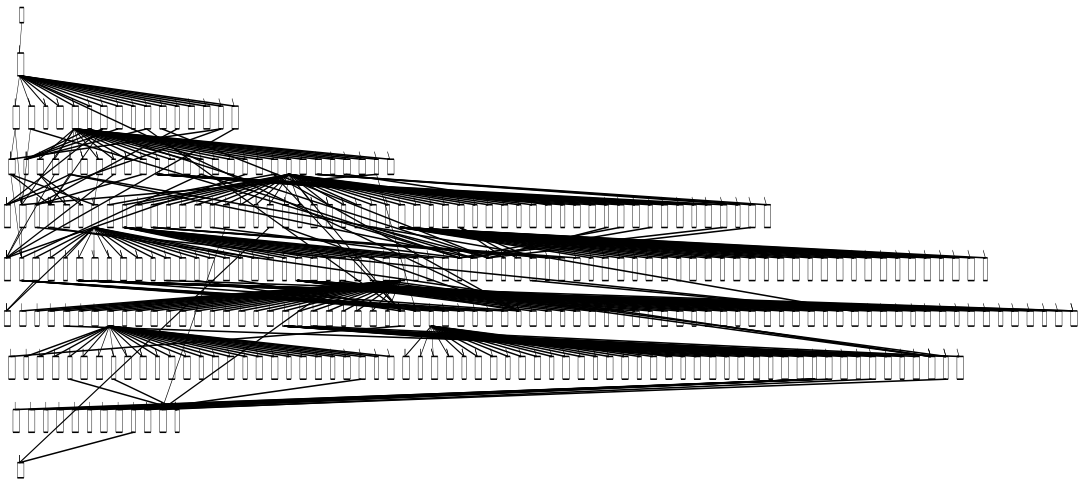

Figure B.2.: Task Graph generated by TGFF with a medium communication pattern.

B.3.3. Dense Communication

File: dense_communication.tgffopt

```
tg_cnt 1
task_cnt 300 1
task_degree 30 30
tg_write
eps_write

table_label COMMUN
table_cnt 1
table_attrib price 80 20
type_attrib exec_time 50 20
trans_write

table_label EXEC
table_cnt 1
pe_write
```

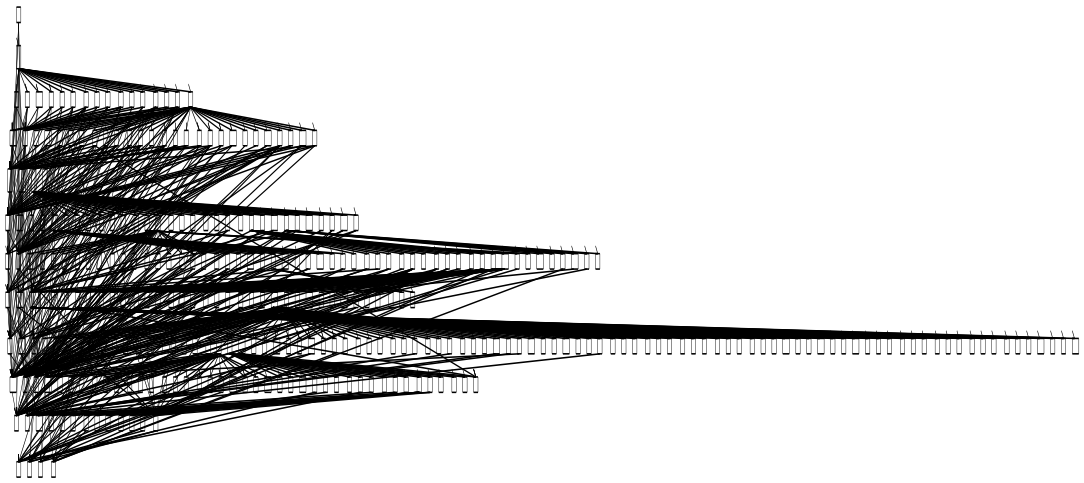

Figure B.3.: Task Graph generated by TGFF with a dense communication pattern.

C. Many-Core Hardware Demonstrator Platform

A prototyping platform consisting of several Atmel 8-bit ATXmega micro-controllers was developed for demonstrating the internal state and the ongoing communications on a real distributed system [SKH11]. By using four serial interfaces per core, a 2d-mesh network is realized. The platform is based on modules consisting of four cores that can be stacked to large system sizes with up to 65280 (255x256) cores (see Figure C.1 and Figure C.2).

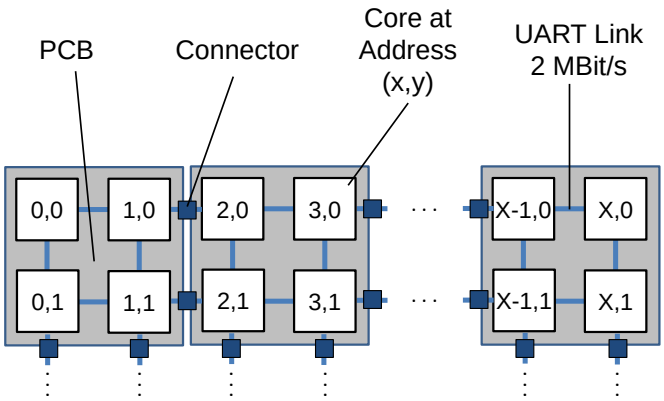

Figure C.1.: Architecture of the demonstrator platform

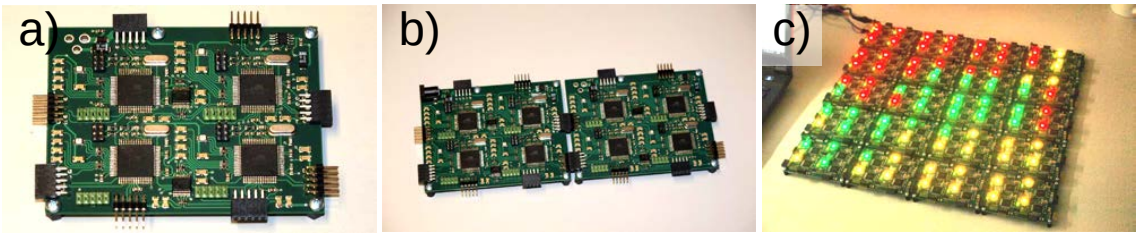

Figure C.2.: Photos of a) a module consisting of four nodes, b) two modules stacked, and c) the Many-Core demonstration platform running AStra on 80 cores. The platform underlines the feasibility of the proposed approach but was **not** used to obtain any of the presented results!

The firmware [SKH12] running on each micro-controller emulates a Network-on-Chip by providing a reliable end-to end communication across the platform. It allows

to execute multiple, independent, concurrent jobs at the same time. Different Agent strategies are implemented. Due to the modular design of the firmware, it is easy to extend the system by implementing new tasks and agents. The firmware allows to re-program the application and Agent implementation across the platform without the need to connect each individual micro-controller to the host-PC.

Due to the limitations of the platform (e.g. the micro-controllers handle the NoC-like communication and message routing in software) it is not suitable for performance measurements (this is far better accomplished in the simulation). No results presented in this thesis are obtained by using the prototyping platform. However, it is suitable for indicating a) the feasibility of a fully distributed resource management and b) the low resource demands of AStra. Table C.1 shows the static memory footprint of the implementation and the used lines of C code for the different components running on the platform, configured to be able to support 100 cores [SKH12]. With growing system sizes, the static RAM demand for the Agent-Base grows with 1 byte per additional core. The Agents additionally use dynamic memory for request handling.

Table C.1.: Static Memory Footprint (Bytes) and #Lines of Code used for the implementation of the presented resource management and infrastructure

	Flash	RAM	LoC
Firmware	7196	3376	1812
Middleware	4976	624	1106
Agent-Base	2944	134	533
DistRM	1344	-	227
Low-Effort	880	-	161
AStra	180	12	39

Hardware Profile "Many-Core Demonstrator"

Number of cores:	4 – 65280 (255x256)
Core architecture:	8Bit AVR
Core clock speed:	32 MHz
Cores per cache coherent domain:	1
Primary means of communication:	Message-Passing
Communication infrastructure topology:	2d-Mesh Network
Communication infrastructure bandwidth:	2 MBit/s per link 4 links per core
On-Chip memory (per core):	16 kByte
Operating system support:	Custom [SKH12]

D. Implementation on the Intel SCC

In [PKH12], DistRM [KBL+11] was implemented on the Intel SCC (see Section 2.1.2 and [HDH+10]). The implementation confirmed the simulation results on a real many-core system. To further improve the accuracy of the many-core system simulation, the performance and latencies of the Intel SCC communication libraries (iRCCE, RCCE, RCKMPI, and Linux TCP/IP-Sockets) were evaluated and compared. The individual algorithms used by DistRM were implemented and benchmarked individually.

To compose a full implementation of DistRM, a component based middleware that abstracts form the Intel SCC's communication libraries and offers a more suitable communication interface to the resource management approach specific software components was created. The respective software architecture is shown in Figure D.1. However, the implementation evaluation has shown that the Intel SCC running one Linux Kernel instance per core does not provide an optimal evaluation environment due to the high latencies and variance in the measurements.

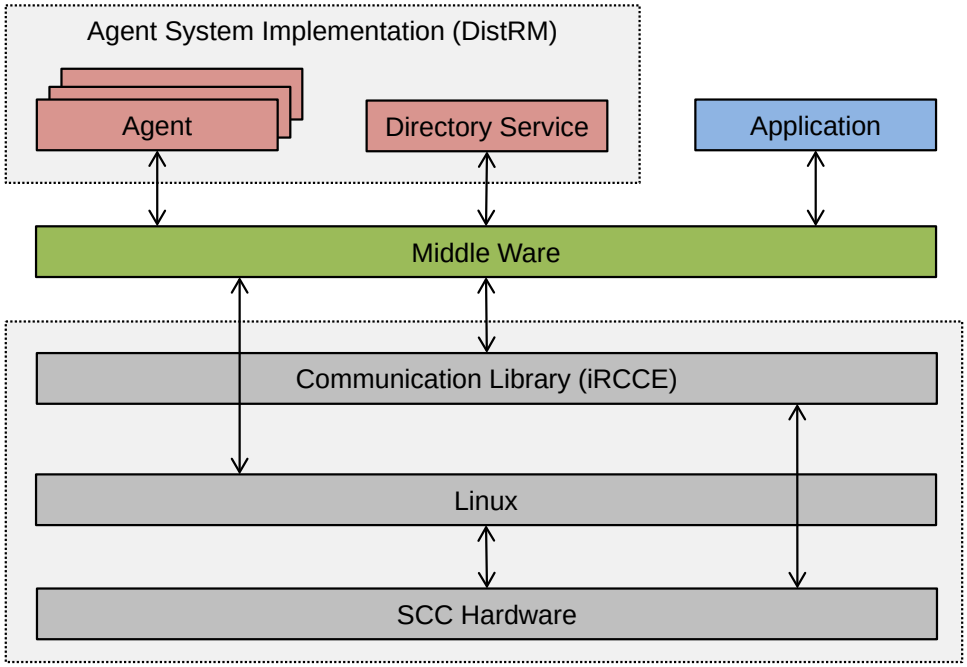

Figure D.1.: Implementation of DistRM on the Intel SCC

Bibliography

[ABD+03] D. H. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, et al. Dynamically tuning processor resources with adaptive processing. *IEEE Computer, Special Issue on Power-Aware Computing*, 36:49–58, 2003.

[AFKH08] M. A. Al Faruque, R. Krist, and J. Henkel. ADAM: run-time agent-based distributed application mapping for on-chip communication. In *Design Automation Conference (DAC)*, pages 760–765, 2008.

[AMCJ+12] M. Alvarez-Mesa, C. C. Chi, B. Juurlink, V. George, and T. Schierl. Parallel video decoding in the emerging HEVC standard. In *Acoustics, Speech and Signal Processing (ICASSP)*, pages 1545–1548. IEEE, 2012.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference (WJCC)*, pages 483–485. ACM, 1967.

[ANF03] K. Aida, W. Natsume, and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 156–163, May 2003.

[Ata98] M. J. Atallah. *Algorithms and theory of computation handbook*. CRC press, 1998.

[ATBS13] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris. Distributed run-time resource management for malleable applications on many-core platforms. In *Design Automation Conference (DAC)*, pages 168–174, 2013.

[AW+07] K. R. Apt, M. Wallace, et al. *Constraint logic programming using ECLiPSe*. Cambridge University Press New York, 2007.

[BABP06] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Design Automation and Test in Europe Conference (DATE)*, pages 15–20, 2006.

[BAG00] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *The Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, volume 1, pages 283–289. IEEE, 2000.

[Bai57] N. T. J. Bailey. *The mathematical theory of epidemics*. 1957.

[Bar14] The barrelfish operating system website. `http://www.barrelfish.org/`, 2014.

[BBD+09] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[BDM02] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[BFH03] F. Berman, G. Fox, and T. Hey. *Grid Computing - Making the Global Infrastructure a Reality*. John Wiley & Sons, Ltd., Chichester, UK, 2003.

[BHK+00] J. Bammi, E. Harcourt, W. Kruitzer, L. Lavagno, and M. Lazarescu. Software performance estimation strategies in a system-level design tool. In *International Workshop on Hardware/Software Codesign (CODES)*, pages 82–86, 2000.

[BIM08] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *International Symposium on Microarchitecture (MICRO)*, pages 318–329, 2008.

[Bir07] K. Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5):8–13, 2007.

[BK90] J. Boillat and P. Kropf. A fast distributed mapping algorithm. In *Conference on Vector and Parallel Processing (CONPAR 90 - VAPP IV)*, volume 457, pages 405–416. 1990.

[BKH14] M. Bertolutti, S. Kobbe, and J. Henkel. Cycle-Accurate Evaluation of Multi-Agent Resource Management in Distributed Many-Core Systems. Bachelor Thesis, Karlsruhe Institute of Technology, 2014.

[BKS00] K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design & Test of Computers*, 17(1):68–83, 2000.

[BL94] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[Bor07] S. Borkar. Thousand core chips: a technology perspective. In *Design Automation Conference (DAC)*, pages 746–749, 2007.

[BSKH07] L. Bauer, M. Shafique, S. Kramer, and J. Henkel. Rispp: rotating instruction set processing platform. In *Design Automation Conference (DAC)*, pages 791–796. ACM, 2007.

[BSKH08] L. Bauer, M. Shafique, S. Kreutz, and J. Henkel. Run-time system for an extensible embedded processor with dynamic instruction set. In *Design Automation and Test in Europe Conference (DATE)*, pages 752–757. IEEE, 2008.

[BWC+03] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[BWCM+10] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. 2010.

[CB00] W. Cirne and F. Berman. Adaptive selection of partition size for supercomputer requests. In *Job Scheduling Strategies for Parallel Processing*, pages 187–207, 2000.

[CBC+10] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Resource management in the tessellation manycore OS. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[CCD+08] Y.-K. Chen, J. Chhugani, P. Dubey, C. Hughes, D. Kim, S. Kumar, V. Lee, A. Nguyen, and M. Smelyanskiy. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE*, 96(5):790–807, 2008.

[CCK12] P. Choudhury, P. Chakrabarti, and R. Kumar. Online scheduling of dynamic task graphs with communication and contention for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):126–133, Jan 2012.

[CCS+08] J. Cheng, J. Castrillon, W. Sheng, H. Sharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *Design Automation Conference (DAC)*, 2008.

[CE12] G. Chrysos and S. P. Engineer. Intel® Xeon Phi™ coprocessor (codename Knights Corner). In *24th Hot Chips Symposium (HC)*, 2012.

[CEH+13] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz. Tessellation: Refactoring the os around explicit resource containers with continuous adaptation. In *Design Automation Conference (DAC)*, pages 76:1–76:10, 2013.

[CFK+98] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer, 1998.

[CGJ78] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[CJS+02] J. Cao, S. A. Jarvis, S. Saini, D. J. Kerbyson, and G. R. Nudd. Arms: An agent-based resource management system for grid computing. *Sci. Program.*, 10:135–148, 2002.

[CMBAN08] M. Curtis-Maury, F. Blagojevic, C. Antonopoulos, and D. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10), 2008.

[DA06] F. Dong and S. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical Report 2006-504, School of Computing, Queen's University, Kingston, Ontario, 2006.

[dDAB+13] B. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. de Massas, F. Jacquet, S. Jones, N. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2013.

[dDdML+13] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18(0):1654 – 1663, 2013. 2013 International Conference on Computational Science.

[DKH13] J. Dobrinkat, S. Kobbe, and J. Henkel. Simulative Design Space Exploration of Resource Management Approaches for Many-Core Systems. Diplomarbeit, Karlsruhe Institute of Technology, 2013.

[DL89] J. Du and J. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, 1989.

[Dow98] A. B. Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, 1(1):133–145, 1998.

[DRW98] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *6th international workshop on Hardware/software codesign (CODES/CASHE)*, pages 97–101, 1998.

[EAYY95] G. Edjlali, G. Agrawal, A. S. Y, and J. S. Y. Data parallel programming in an adaptive environment. In *International Parallel Processing Symposium*, 1995.

[ECEP06] C. Erbas, S. Cerav-Erbas, and A. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374, 2006.

[EFH09] T. Ebi, A. Faruque, and J. Henkel. Tape: Thermal-aware agent-based power econom multi/many-core architectures. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 302–309. IEEE, 2009.

[EZL89] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38:408–423, 1989.

[FAEH07] A. Faruque, M. Abdullah, T. Ebi, and J. Henkel. Run-time adaptive on-chip communication scheme. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 26–31, 2007.

[FAEH08] A. Faruque, M. Abdullah, T. Ebi, and J. Henkel. Roadnoc: Runtime observability for an adaptive network on chip architecture. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 543–548, 2008.

[Fei03] D. Feitelson. Metric and workload effects on computer systems evaluation. *Computer*, 36(9):18–25, 2003.

[Fei05] D. Feitelson. Parallel Workloads Archive, 2005.

[FFK+97] S. Fitzgerald, I. Foster, C. Kesselman, G. Von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *IEEE International Symposium on High Performance Distributed Computing*, pages 365–375, 1997.

[FK03] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[Fos01] I. Foster. The globus toolkit for grid computing. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 2–2, 2001.

[Fou] T. L. Foundation. The linux kernel archives. `http://www.kernel.org/`.

[FR96] D. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[FR98] D. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer, 1998.

[FRS+97] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 1291, pages 1–34. 1997.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.

[Ghu07] A. Ghuloum. Ct: channelling NeSL and SISAL in C++. In *ACM SIGPLAN workshop on Commercial users of functional programming (CUFP)*, 2007.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[Gus88] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[HBB+11] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Hartig, L. Hedrich, et al. Design and architectures for dependable embedded systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 69–78. IEEE, 2011.

[HBH12] F. Hameed, L. Bauer, and J. Henkel. Dynamic cache management in multi-core architectures through run-time adaptation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 485–490. IEEE, 2012.

[HDH+10] 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 *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108–109, 2010.

[HES+10] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *7th*

International Conference on Autonomic Computing, ICAC '10, pages 79–88, 2010.

[HHB+12] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive manycore architectures. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 193–200, 2012.

[HLK04] C. Huang, O. Lawlor, and L. V. Kale. Adaptive MPI. In *Languages and Compilers for Parallel Computing*, pages 306–322. Springer, 2004.

[HM98] M. W. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. *Concurrency: Practice and Experience*, 10(14), 1998.

[HMS+11] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. SEEC: A framework for self-aware management of multicore resources. Technical Report MIT-CSAIL-TR-2011-016, Computer Science and Artificial Intelligence Laboratory, MIT, March 2011.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[Hor01] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. *Computing Systems*, 2007(Jan):1–40, 2001.

[HSW01] J. Hungershöfer, A. Streit, and J.-M. Wierum. Efficient resource management for malleable applications, 2001.

[HZKK06] C. Huang, G. Zheng, L. Kalé, and S. Kumar. Performance evaluation of adaptive mpi. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 12–21. ACM, 2006.

[HZQ+13] T. Huang, Y. Zhu, M. Qiu, X. Yin, and X. Wang. Extending Amdahl's law and Gustafson's law by evaluating interconnections on multi-core processors. *The Journal of Supercomputing*, 66(1):305–319, 2013.

[HZZ+14] J. Heisswolf, A. Zaib, A. Zwinkau, S. Kobbe, A. Weichslgartner, J. Teich, J. Henkel, G. Snelting, , A. Herkersdorf, and J. Becker. Cap: Communication aware programming. In *Design Automation Conference (DAC)*, 2014.

[IDSSM05] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Parallel Processing (Euro-Par)*, pages 196–205. 2005.

[IKLL12] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[Int14] Intel. Threading Building Blocks (Intel® TBB). `http://www.threadingbuildingblocks.org/`, 2014.

[IPS96] N. Islam, A. Prodromidis, and M. S. Squillante. Dynamic partitioning in different distributed-memory environments. In *Job Scheduling Strategies for Parallel Processing*, 1996.

[Ism04] M. Ismail. Parallel genetic algorithms (pgas): master slave paradigm approach using mpi. In *E-Tech 2004*, pages 83–87, July 2004.

[ITR13] ITRS. International technology roadmap for semiconductors. `http://www.itrs.net/Links/2013ITRS/Home2013.htm`, 2013.

[JAFH11] J. Jahn, M. Al Faruque, and J. Henkel. CARAT: Context-aware runtime adaptive task migration for multi core architectures. In *Design Automation and Test in Europe Conference (DATE)*, pages 515–520, 2011.

[JKP+12] J. Jahn, S. Kobbe, S. Pagani, J.-J. Chen, and J. Henkel. Work in Progress: Malleable Software Pipelines for Efficient Many-core System Utilization. In E. Noulard and S. Vernhes, editors, *Many-core Applications Research Community (MARC) Symposium*, pages 30–33, Toulouse, France, 2012. ONERA, The French Aerospace Lab.

[JMB05] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.

[JPCH13] J. Jahn, S. Pagani, J.-J. Chen, and J. Henkel. Moma: Mapping of memory-intensive software-pipelined applications for systems with multiple memory controllers. In *Computer-Aided Design (ICCAD)*, pages 508–515. IEEE, 2013.

[JPK+13] J. Jahn, S. Pagani, S. Kobbe, J.-J. Chen, and J. Henkel. Optimizations for configuring and mapping software pipelines in many core. In *Design Automation Conference (DAC)*, pages 130:1–130:8, 2013.

[JSCT08] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *International conference on Parallel architectures and compilation techniques (PACT)*, pages 220–229, 2008.

[JSW98] N. R. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.

[KAB+00] R. A. Kendall, E. Apra, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. Straatsma, T. L. Windus, and A. T. Wong". High performance computational chemistry: An overview of NWChem a distributed parallel application. *Computer Physics Communications*, 128:260–283, 2000.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing Congress (IFIP)*, volume 74, pages 471–475, 1974.

[KBH15] S. Kobbe, L. Bauer, and J. Henkel. Adaptive on-the-fly application performance modeling for many cores. In *Design Automation and Test in Europe Conference (DATE)*, Mar 2015.

[KBL+11] S. Kobbe, L. Bauer, D. Lohman, W. Schröder-Preikschat, and J. Henkel. DistRM: Distributed resource management for on-chip many-core systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, 2011.

[KBL+14] S. Kobbe, L. Bauer, D. Lohman, W. Schröder-Preikschat, and J. Henkel. AStra: Adaptive strategy selection for distributed resource management in many-core systems. *ACM Transactions on Parallel Computing (TOPC)*, pages 1–30, 2014. Submitted for Review.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KK93] L. V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Conference on Object Oriented Programming. Systems, Languages and Applications (OOPSLA)*, 1993.

[KKD02] L. V. Kalé, S. Kumar, and J. Desouza. A malleable-job system for timeshared parallel machines. In *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2002.

[KKD04] D. Kempe, J. Kleinberg, and A. Demers. Spatial gossip and resource location protocols. *Journal of the ACM (JACM)*, 51(6):943–967, 2004.

[KKG+11] G. Kousiouris, D. Kyriazis, S. Gogouvitis, A. Menychtas, K. Konstanteli, and T. Varvarigou. Translation of application-level terms to resource-level attributes across the cloud stack layers. In *Computers and Communications (ISCC)*, pages 153–160. IEEE, 2011.

[KKH13] R. Klöpfer, S. Kobbe, and J. Henkel. Taskgraph basiertes Anwendungsmodell mit heuristischer Erkennung von charakteristischen Anwendungsphasen. Studienarbeit, Karlsruhe Institute of Technology, 2013.

[KKH14] K. S. Klousseh, S. Kobbe, and J. Henkel. Analytical Resource-Aware Multi-Application Mapping to Many-Cores. Diplomarbeit, Karlsruhe Institute of Technology, 2014.

[KKJC02] D.-W. Kim, K.-H. Kim, W. Jang, and F. F. Chen. Unrelated parallel machine scheduling with setup times using simulated annealing. *Robotics and Computer-Integrated Manufacturing*, 18(3-4):223–231, 2002.

[KMP+10] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. ATAC: a 1000-core cache-coherent processor with on-chip optical network. In *19th international conference on Parallel architectures and compilation techniques*, pages 477–488, 2010.

[Lie95] J. Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

[LK03] T. Lei and S. Kumar. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *Euromicro Symposium on Digital System Design*, pages 180–187, 2003.

[LKB+09] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, pages 10–10, 2009.

[LNC13] X. Liang, M. Nguyen, and H. Che. Wimpy or brawny cores: A throughput perspective. *Journal of Parallel and Distributed Computing*, 73(10):1351–1361, 2013.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[LRC+11] M. Lis, P. Ren, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, and S. Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 175–185. IEEE, 2011.

[M+65] G. E. Moore et al. Cramming more components onto integrated circuits, 1965.

[May] D. May. David may's transputer page. `http://www.cs.bris.ac.uk/~dave/transputer.html`.

[May83] D. May. Occam. *SIGPLAN Not.*, 18(4):69–79, April 1983.

[MBS+05] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner. Time and energy efficient mapping of embedded applications onto NoCs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, volume 1, pages 33–38, January 2005.

[MHS+10] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *49th IEEE Conference on Decision and Control (CDC)*, pages 3736–3741, 2010.

[MMCM07] C. Marcon, E. Moreno, N. Calazans, and F. Moraes. Evaluation of algorithms for low energy mapping onto NoCs. In *International Symposium on Circuits and Systems (ISCAS)*, pages 389–392, 2007.

[MNMN97] J. E. Moreira, V. K. Naik, J. E. Moreira, and V. K. Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41, 1997.

[MRL+10] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: The programmer's view. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, 2010.

[NO09] A. Nedic and A. Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1):48–61, 2009.

[OPF09] S. Ostermann, R. Prodan, and T. Fahringer. Extending grids with cloud resource management for scientific computing. In *IEEE/ACM International Conference on Grid Computing*, pages 42–49, 2009.

[ORSA05] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *International Symposium on Microarchitecture (MICRO)*, 2005.

[OSK+11] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. OctoPOS: A parallel operating system for invasive computing. In *International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, pages 9–14, 2011.

[Ous82] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *International Conference on Distributed Computing Systems (ICDCS)*, volume 82, pages 22–30, 1982.

[Pet12] S. Peter. *Resource Management in a Multicore Operating System*. PhD thesis, Carl-von-Ossietzky Universität Oldenburg, 2012.

[Phe08] C. Pheatt. Intel®threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[PKH12] D. Pahl, S. Kobbe, and J. Henkel. Distributed Resource Management on the Intel Single-Chip Cloud Computer. Diplomarbeit, Karlsruhe Institute of Technology, 2012.

[PLMS00] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[PRB10] S. Peter, T. Roscoe, and A. Baumann. Barrelfish on the intel single-chip cloud computer, barrelfish technical note 005. Technical report, Tech. Rep., 2010., 2010.

[PSB+10] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe. Design principles for end-to-end multicore schedulers. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[PSK+12] J. Paul, W. Stechele, M. Krohnert, T. Asfour, and R. Dillmann. Invasive computing for robotic vision. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 207–212, Jan 2012.

[PYL13] J.-S. Pan, C.-N. Yang, and C.-C. Lin. Effective processor allocation for moldable jobs with application speedup model. In *Advances in Intelligent Systems and Applications - Volume 2*, volume 21 of *Smart Innovation, Systems and Technologies*, pages 563–572. 2013.

[Rei07] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.", 2007.

[RHW88] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 1988.

[RNV09] S. S. Ram, A. Nedic, and V. V. Veeravalli. Asynchronous gossip algorithms for stochastic optimization. In *IEEE Conference on Decision and Control*, pages 3581–3586, 2009.

[SATG+07] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP environment. *ACM SIGOPS Operating Systems Review*, 41(3):73–86, 2007.

[SBHH15] F. Samie, L. Bauer, C.-M. Hsieh, and J. Henkel. Online binding of applications to multiple clock domains in shared fpga-based systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 25–30, 2015.

[SBR+12] L. Schor, I. Bacivarov, D. Rai, H. Yang, S. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 71–80, 2012.

[SCH11] H. Sun, Y. Cao, and W.-J. Hsu. Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):594–607, April 2011.

[SGB+09] H. Shojaei, A.-H. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management. In *Design Automation Conference (DAC)*, pages 917–922, 2009.

[Sha07] J. Shalf. The new landscape of parallel computer architecture. In *Journal of Physics: Conference Series*, volume 78, page 012066. IOP Publishing, 2007.

[Sil14] Silicon Graphics International Corp. *SGI UV: Big Brain for No-Limit Computing*, 2014. `https://www.sgi.com/products/servers/uv/`.

[SKH11] Y. Song, S. Kobbe, and J. Henkel. Hardware Emulation Platform for Future On-Chip Many-Core Systems. Diplomarbeit, Karlsruhe Institute of Technology, 2011.

[SKH12] F. T. Schandinat, S. Kobbe, and J. Henkel. Distributed Resource Management on a On-Chip Many-Core System Demonstration Platform. Bachelor Thesis, Karlsruhe Institute of Technology, 2012.

[SLS07] G. Sabin, M. Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In *Job Scheduling Strategies for Parallel Processing*, volume 4376, pages 94–114. 2007.

[Smi80] R. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, 29:12, 1980.

[SS11] P. Sanders and J. Speck. Efficient parallel scheduling of malleable tasks. In *Parallel Distributed Processing Symposium (IPDPS), IEEE International*, pages 1156–1166, 2011.

[SSKH13] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Design Automation Conference (DAC)*, 2013.

[SV96] S. Sahni and G. Vairaktarakis. The master-slave paradigm in parallel computer and industrial settings. *Journal of Global Optimization*, 9(3-4):357–377, 1996.

[TCA07] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[tes14] Tessellation OS website. `http://tessellation.cs.berkeley.edu/`, 2014.

[THH+11] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, pages 241–268. Springer, 2011.

[THW02] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[Til14] Tilera Corporation. Tile-GX Processor Family: Whitepaper, 2014.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[TKH11] J. P. Tsague, S. Kobbe, and J. Henkel. Untersuchung von optimalen Schedules für variabel parallele Anwendungen. Studienarbeit, Karlsruhe Institute of Technology, 2011.

[TWY92] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Symposium on Parallel algorithms and architectures (SPAA)*, pages 323–332, 1992.

[UCL04] G. Utrera, J. Corbalan, and J. Labarta. Implementing malleability on MPI jobs. In *13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 215–224, 2004.

[VBC11] A. Vajda, M. Brorsson, and D. Corcoran. *Programming many-core chips*. Springer, 2011.

[VBG+10] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, et al. NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[VD02] S. S. Vadhiyar and J. J. Dongarra. Srs - a framework for developing malleable and migratable parallel applications for distributed systems. In *Parallel Processing Letters*, 2002.

[Wei99] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[WS85] C. Whitby-Strevens. The transputer. In *International Symposium on Computer Architecture*, ISCA '85, pages 292–300, 1985.

[WZT13] S. Wildermann, T. Ziermann, and J. Teich. Game-theoretic analysis of decentralized core allocation schemes on many-core systems. In *Design Automation and Test in Europe Conference (DATE)*, pages 1498–1503, 2013.

[YGSP13] B. Yang, L. Guang, T. Säntti, and J. Plosila. Mapping multiple applications with unbounded and bounded number of cores on many-core networks-on-chip. *Microprocessors and Microsystems*, 37(4):460–471, 2013.

[YGX+10] B. Yang, L. Guang, T. Xu, A. Yin, T. Säntti, and J. Plosila. Multi-application multi-step mapping method for many-core Network-on-Chips. In *NORCHIP, 2010*, pages 1–6, Nov 2010.

[YXSP10] B. Yang, T. Xu, T. Säntti, and J. Plosila. Tree-model based mapping for energy-efficient and low-latency Network-on-Chip. In *13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 189–192, April 2010.

[ZA10] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *International Symposium on High Performance Distributed Computing (HPDC)*, pages 304–307, 2010.

[ZWB00] B. Zhou, D. Walsh, and R. Brent. Resource allocation schemes for gang scheduling. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lecture Notes in Computer Science*, pages 74–86. Springer Berlin / Heidelberg, 2000.

List of Figures

List of Tables

List of Listings

List of Symbols

A_i Application 'i'.

C_{A_i} Set of cores $\{c_a, c_b, \ldots\}$ allocated to Application 'i'.

$|C_x|$ Number of cores in set C_x.

$\mathcal{C}$ Set of of all cores $\{c_1, c_2, \ldots, c_N\}$ in the many-core system.

c_a Core 'a' at topological location (c_a^x, c_a^y).

c_a^x, c_a^y Topological location of core c_a defined by its x and y coordinates in the 2d-mesh network.

$E_{A_i,P_j}(C_{A_i})$ Execution efficiency (speedup per core) of application A_i in phase P_j when executed on the set of cores C_{A_i}.

$E^{\#}_{A_i,P_j}(k)$ Execution efficiency (speedup per core) of application A_i in phase P_j when executed on k cores.

$T^{make}_{A_1,A_2,\ldots,A_M}$ Point in time when all applications $\{A_1, A_2, \ldots, A_M\}$ finished computation.

$w(c_a, c_b)$ Number of hops in the 2d-mesh network between the cores c_a and c_b, also referred to as the Manhattan-distance.

$w_{avg}(C_x)$ Average number of hops in the 2d-mesh network between the cores in C_x.

$w^{\#,max}_{avg}(k)$ Maximal (worst-case) average number of hops between cores when selecting k out of N cores.

$w^{\#,min}_{avg}(k)$ Minimal (best-case) average number of hops between cores when selecting k out of N cores.

$N^{\Phi_{max}}_{A_i}$ Number of cores that maximizes the speedup multiplied by the efficiency of A_i.

P_j Phase 'j'. Typically F_{A_i,P_j} refers to any function F that represents the phase P_j of application A_i.

p_{A_i} Relative progress of application A_i, when $p_{A_i} = 1$ the application completed its workload.

$Q^{goal}_{A_i}(C_{A_i})$ Mapping quality of application A_i when using the cores in C_{A_i} with respect to the targeted optimization *goal*.

$Q^{goal}(A_1, A_2, \ldots, A_M)$ Overall application mapping quality with respect to the targeted optimization *goal*.

$S_{A_i,P_j}(C_{A_i})$ Relative execution time of application A_i in phase P_j when executed on C_{A_i} compared to execution on only one core.

$S^{average}_{A_1,A_2,\ldots,A_M}$ Average speedup of the applications $\{A_1, A_2, \ldots, A_M\}$.

$S^{\#}_{A_i,P_j}(k)$ Relative execution time of application A_i in phase P_j when executed on k cores compared to execution on only one core.

S_{A_i} Speedup function of A_i, used interchangeably with the parameter tuple $\left(P^{best}_{A_i}, \sigma^{best}_{A_i}, P^{worst}_{A_i}, \sigma^{worst}_{A_i}\right)$.

$S^{\#,best}_{A_i}(k)$ Speedup function of A_i to express the best-case speedup when using k cores spatially close to each other.

$S^{\#,worst}_{A_i}(k)$ Speedup function of A_i to express the worst-case speedup when using k cores spatially as distant as possible from each other.

M Number of concurrently executing applications.

N Number of cores in the many-core system.

$T^{finish}_{A_i}(C_{A_i})$ Point in time A_i finishes computation when using the resources in C_{A_i}.

T_{A_i} Total runtime of application A_i when using only one core.

$T^{ready}_{A_i}$ Point in time A_i is ready for execution.

$T_{A_i}(C_{A_i})$ Time A_i required to finish computation while using the resources in C_{A_i}.

$T^{\#}_{A_i}(k)$ Time A_i required to finish computation while using k cores.

$U_{A_i}(C_{A_i})$ Function that determines the utility of the cores in C_{A_i} to A_i.

$U^{\#}_{A_i}(k)$ Function that determines the utility of k cores to A_i without considering their topological location.

$U^{goal}_{A_i}(k)$ Function that determines the utility of the cores in C_{A_i} to A_i with respect to the specified optimization *goal*.