Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Market-Based Resourse Management for Many-Core Systems

Διπλωματικη Εργασια

του

ΘΕΜΙΣΤΟΚΛΗ Γ. ΜΕΛΙΣΣΑΡΗ

Επιβλέπων: Δημήτριος Σούντρης
Επ. Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων & Ψηφιακων Συστηματων
Αθήνα, Ιούλιος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Market-Based Resourse Management for Many-Core Systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΘΕΜΙΣΤΟΚΛΗ Γ. ΜΕΛΙΣΣΑΡΗ

Επιβλέπων: Δημήτριος Σούντρης
Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την η Ιουλίου 2014.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Δημήτριος Σούντρης	Κιαμάλ Ζ. Πεκμεστζή	Γεώργιος Οικονομάκος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2014

(Υπογραφή)

....

ΘΕΜΙΣΤΟΚΛΗΣ Γ. ΜΕΛΙΣΣΑΡΗΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Ευχαριστίες

Θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες προς τον επιβλέποντα καθηγητή κ. Δ. Σούντρη για την εμπιστοσύνη και την πολύτιμη καθοδήγηση καθ΄ όλη τη διάρκεια της άριστης αυτής συνεργασίας, που οδήγησαν στην επιτυχημένη ολοκλήρωση της διπλωματικής εργασίας. Επίσης, ευχαριστώ ιδιαίτερα τον διδάκτορα Ηρακλή Αναγνωστόπουλο, η συμβολή του οποίου με τις γνώσεις, την πολύτιμη βοήθεια και την συνεχή υποστήριξη ήταν καθοριστική στην πορεία της εργασίας. Τέλος, θα ήθελα να ευχαριστήσω τον διδάκτορα Αλέξανδρο Μπάρτζα που βοήθησε σημαντικά με τις συμβουλές του.

Περίληψη

Αντικείμενο της διπλωματικής αποτελεί η μελέτη και η ανάπτυξη μιας κλιμακώσιμης και κατανεμημένης πλατφόρμας (framework) διαχείρισης πόρων σε χρόνο εκτέλεσης για συστήματα πολλαπλών πυρήνων. Σε αυτήν την πλατφόρμα η διαχείριση πόρων είναι βασισμένη σε μοντέλα, τα οποία είναι εμπνευσμένα από την οικονομία. Παρουσιάζεται ένας διαχειριστής πόρων, ο οποίος προσφέρει ένα περιβάλλον διαχείρισης πόρων και εφαρμογών καθ΄ όλη τη διάρκεια ζωής τους, στο οποίο η κατανομή και δρομολόγηση των εφαρμογών στους πόρους πραγματοποιείται με αλγόριθμους βασισμένους σε κανόνες αγοράς. Η αποδοτικότητα κάθε μοντέλου αξιολογείται βάσει της πτώσης της αξιοπιστίας των πόρων (μετρική MTTF-Mean Time To Failure).

Λέξεις Κλειδιά

Πολυ-πύρηνα Συστήματα, Κατανεμημένη Διαχείριση Πόρων, Διαχείριση πόρων σε χρόνο εκτέλεσης, Οικονομικά μοντέλα

Abstract

The purpose of this diploma thesis is the design and development of a scalable and distributed run-time resource management framework for Many-core systems. In this framework, resource management is based on economy-inspired models. The presented resource management framework offers an environment that manages both application tasks and resources at run-time, matches and distributes application tasks across resources with algorithms which are based on market principles. The efficiency of each model is evaluated with respect to resource reliability degradation (metric MTTF-Mean Time to Failure).

Keywords

Many-core Systems, Distributed Resource Management, Run-time Resource Management, Economic models

Contents

List of Figures

Chapter 1

Background knowledge

1.1 Introduction

According to Gordon Moore's prediction, the number of transistors in an integrated circuit doubles approximately every two years, leading to more complex systems. However, Dennard scaling suggests that power requirements are proportional to area for transistors. Combined with Moore's law, power density grows at about the same rate as transistor density, making integrated circuits unable to scale down in terms of technology at the same rate. In addition, architecture related modifications cause limited improvement and therefore single-core performance is limited. Despite these, processor chip computing power continues to grow by harnessing the additional processing power from additional processor cores (or CPUs), introducing the use of multicore processor chips [21].
The current trend in System on Chip (SoC) design is the shift from multicore processor chips to many-core [10]. This change towards many processor cores is not only a change in scale, but also a change in the kind of the processor chips.

In 2005, the industry started adopting multicore general purpose processors, which

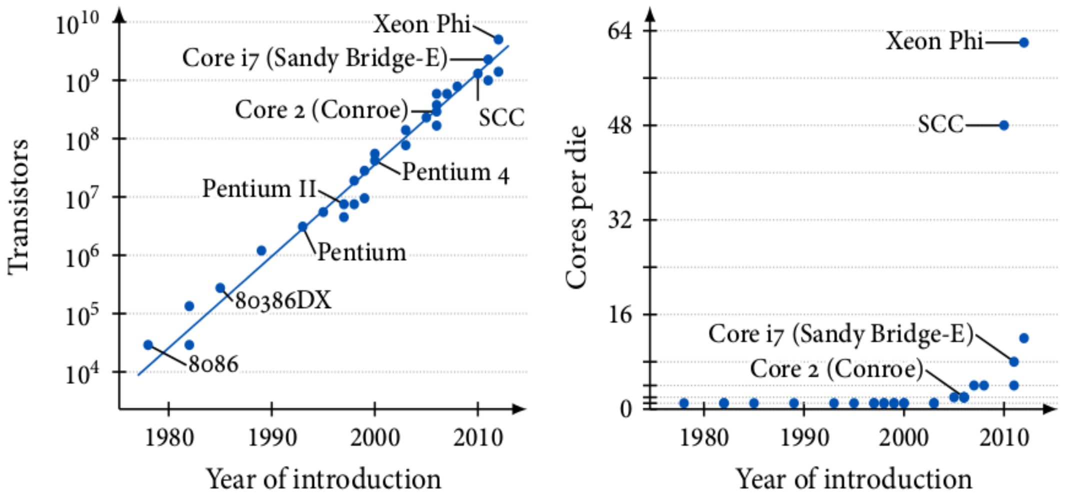

Figure 1.1: Intel Microprocessors

marked the entrance into the multicore era. Since then, the number of cores in a single chip continues to grow, as it is illustrated in figure 1.1 [29]. Previously, systems used parallelism by putting multiple processors and organizing them in servers and supercomputers. Now, modern processors make every computing system able to use parallel computation. Embedded systems and general purpose systems begin to show signs of convergence, both in terms of software and hardware. The industry dictates increased functionality and at the same time, increased performance and responsiveness, encouraging the shift away from single core systems. Initially, mobile phones only featured basic functionalities with the focus on communication and therefore its hardware requirements were minimal. On the contrary, cutting edge mobile phones and more specifically smartphones use up to 8 cores (e.g. Samsung S5 featuring ARM-based Exynos 5 octa) and feature high definition cameras, hardware accelerators and multiple sensors. Their processing capabilities are equivalent to modern general purpose computing systems[31].

Adding multiple cores to a system adds raw processing power. However, this does not imply that the software makes good use of the processing power. Software needs to alleviate problems caused by the nature of many-core systems in order to successfully manage and coordinate the available resources towards creating an efficient system.
Software for a single-core system behaves in a way programmers can understand easily. Instructions are executed in the order that is defined, and if designed carefully, the program always gives a correct result. In great contrast, exploiting hardware parallelism (in the form of multiple cores) by concurrent programming can only be accomplished when the software is changed. The program has to be split in chunks of work that can be executed in parallel. Programmers accept the fact that they have to face programming for concurrency as well as to face difficult bugs related to communication and heterogeneous hardware componennt latency. This truly requires co-design of the hardware architecture and the programming approach.
This complexity in multicore systems makes it difficult for the software to efficiently manage the additional requirements and constraints introduced by the hardware. In Figure 1.2 both hardware and software productivity are represented. In particular, the demand for software is currently doubling every 10 months, the capability of technology is currently doubling every 36 months, as well as the productivity of hardware and software design. Productivity of the hardware designers has been improved recently as systems can be created by integrating multiple cores. Thus, all the functionality is provided only with additional software. Software productivity and especially the one regarding hardware-dependent software, is growing at a much slower rate and doubles only every 5 years. The red arrow illustrates the magnitude of the new design gap including both hardware and software.

1.2 Many-core Systems

1.2.1 Multiple cores

Multi-core processors can be categorised as homogeneous or as heterogenous. Most current general-purpose multi-core processors are homogeneous both in terms of ISA (instruction set architecture) and performance. Homogeneous systems can execute the same binaries and programs on any of their cores without causing any difference in functionality. Many modern multi-core architectures, however provide the capability for system software to dynamically scale the voltage and clock frequency for each core individually. This technique is most commonly referred to as DVFS (Dynamic Voltage and Frequency scaling) and is most often used in embedded processors for power management. The processor can

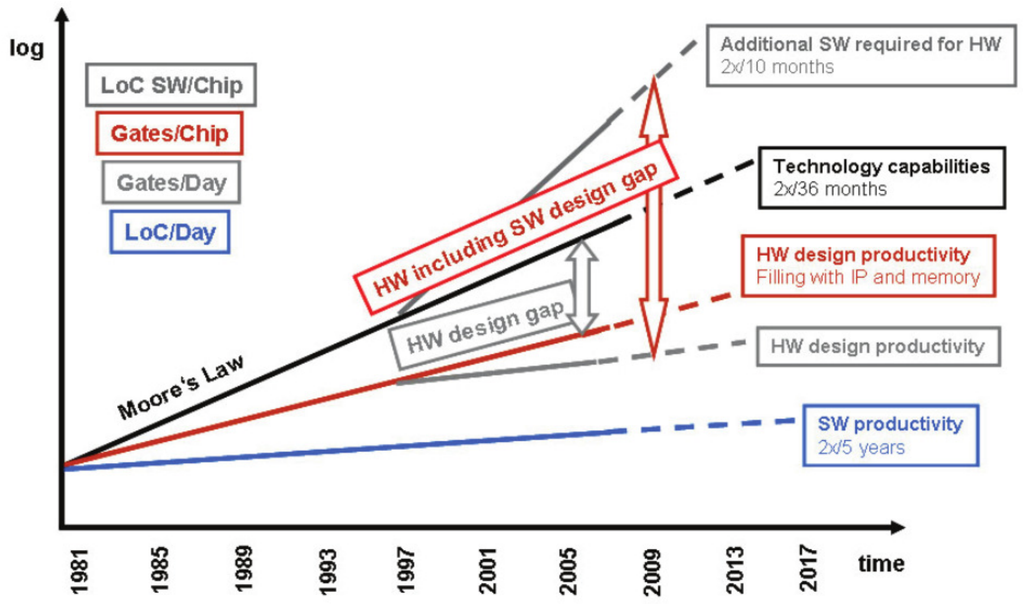

Figure 1.2: Productivity gap between hardware and software design [22]

regulate its voltage and frequency states accordingly in order to either save power or to temporarily increase single-thread performance.

In order for the cores to be indistinguishable at the software level, most of the homogeneous architectures use a shared global address space. The processes or threads can even migrate from one core to the other at run-time, but the cores are viewed as identical.

Unlike the homogeneous processors, multi-core systems which feature at least two different kinds of cores with possible difference in the instruction set architecture (ISA), functionality and performance, are called heterogeneous. Certain application domains that are limited by the efficiency and capability of general purpose processors can still support the design of custom architectures if they are driven by a large enough market. These conditions exist in domains such as computer graphics, signal processing and embedded computing where accelerators have been designed that offer significant performance and efficiency advantages over general purpose architectures [14]. A typical example of a heterogeneous multi-core architecture is the Cell BE architecture (co-developed by IBM, Sony and Toshiba), used in high performance computing applications, such as in gaming devices.

When it comes to programming architectures for parallel computation, a homogeneous architecture with shared global memory is significantly easier to program for than a heterogeneous architecture where the cores do not feature the same instruction set. However, when an application is partitioned into long lived threads with limited or regular communication it is efficient to have some partitions utilize the task-specific cores.

Regarding the core Microarchitecture, great differentiation can be observed. All modern processor cores are pipelined in order to improve on overall throughput by breaking down instruction decoding and instruction execution in stages. By using pipelined cores,

instruction latency is the same or even increased. In addition, designs that aim at high-performance feature speculative dynamic instruction scheduling. These microarchitectural techniques increase the average number of instructions executed per clock cycle. Despite this fact, these tend to become of lower significance in modern multi-core architectures, as they offer limited instruction-level parallelism (ILP), tend to be both complicated and power-hungry and take up valuable silicon surface. As a matter of fact, new architectures such as Intel′s Knights Corner, turn to simple single-issue in-order cores (Knights Corner cores also have powerful vector instructions in order to take up less silicon and reduce power consumption).
In order to compensate for limited instruction level parallelism added to the most advanced cores is simultaneous multithreading (SMT). This hardware technique allows better utilization of hardware resources where a multi-issue pipeline can select instructions from two or more threads. For applications with abundant ILP, single-thread performance is high, while with reduced ILP, thread-level parallelism can be utilized. Simultaneous multithreading is adopted as it introduced only a little additional area and power consumption [33].

1.2.2 Interconnection Networks

In a multi-core setting, there is need to have mechanisms for cores to efficiently communicate with each other. An early approach to achieve inter-core communication in a multicore chip has been through a common shared bus. In order to overstress the bus with memory and I/O traffic, the design includes local cache memories, typically structured in one or two levels between the processor and the bus. The shared bus also enables the implementation of cache coherence protocols(cores are able to broadcast the shared memory operations in the communication medium in their respective order).
More recent designs have abandoned the shared bus as a communication medium as it is not an efficient solution in terms of latency and bandwidth. One reason is that the shared bus has long electrical wires that create delays. Additionally, all the cores of the system are connected to the shared bus, which does not enable an individual core to have all the bandwidth of the bus allocated for itself.
Other popular designs use a crossbar interconnect. The crossbar connects processor modules that include a two level cache hierarchy with the last level cache and memory interfaces. Emerging designs focus on bandwidth and power consumption improvements and include ring busses and switched on chip networks.

1.2.3 Memory

A key target of multi-core systems is to enable parallelization of the software. Therefore, processing cores need an efficient way to communicate with other. While the actual communication is achieved over the interconnection network, cores can store data in shared memory locations and access them afterwards.

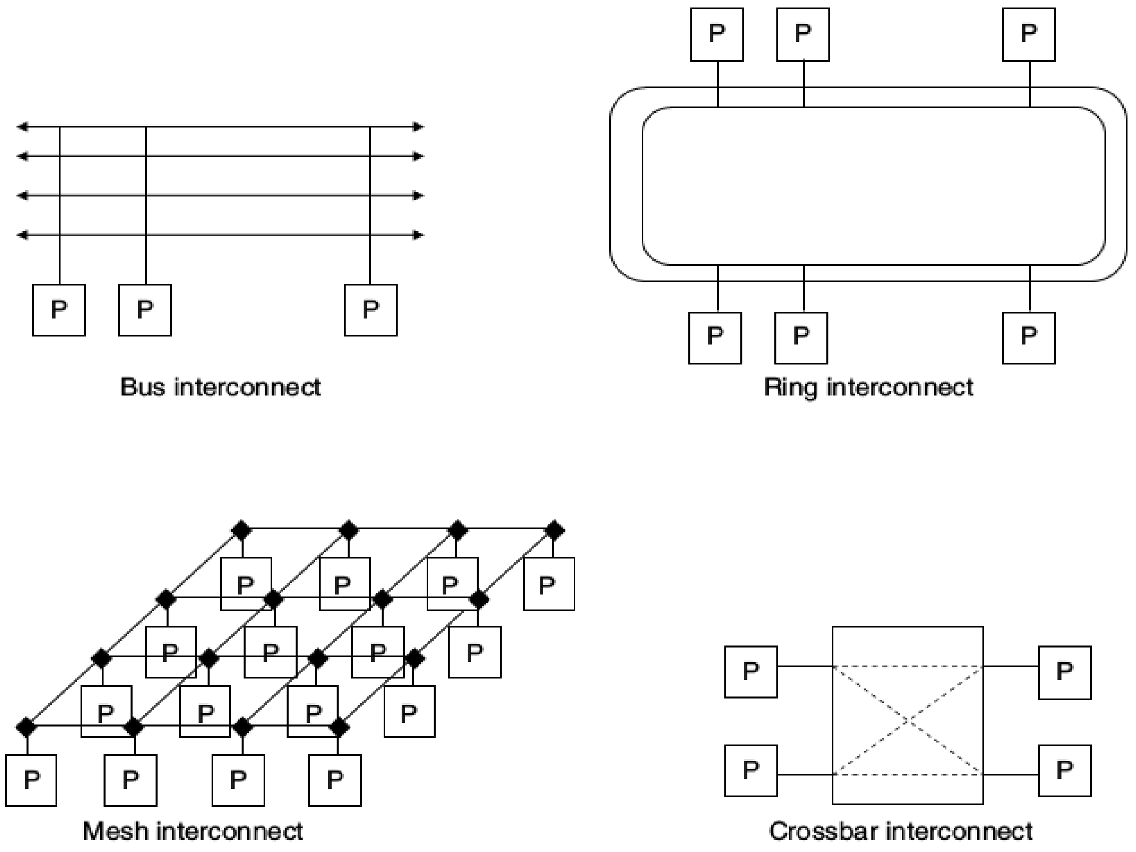

Figure 1.3: Interconnect Architectures [33]

Shared memory is not scalable to thousands of nodes and for this reason it has not been used in high performance computing. Clusters most often use message passing to achieve inter-node communication. Their nodes however, implement a shared address space.
For all these reasons, shared memory has been introduced in all modern general-purpose multi-core processors. In addition, memory systems that implement a shared address space need to have cache-coherence. Cache coherence allows the cores to have local copies of its frequently accessed data in private caches, but also a globally consistent view of the values written to each memory location when multiple cores update shared resources.
Many memory hierarchies have been proposed, based on different cache structures. The memory system can have only shared or only private caches or both and the architectures can be flat or hierarchical. Figure 1.5 displays different cache architectures that have the characteristics mentioned above.

1.2.4 Scalability Considerations for Many-Core Processors

To take full advantage of a many-core system, a high percentage of core utilization is required. In order to facilitate this, a large number of tasks is required and more specifically, tasks that are parallel. Because of this, it is important to coordinate the

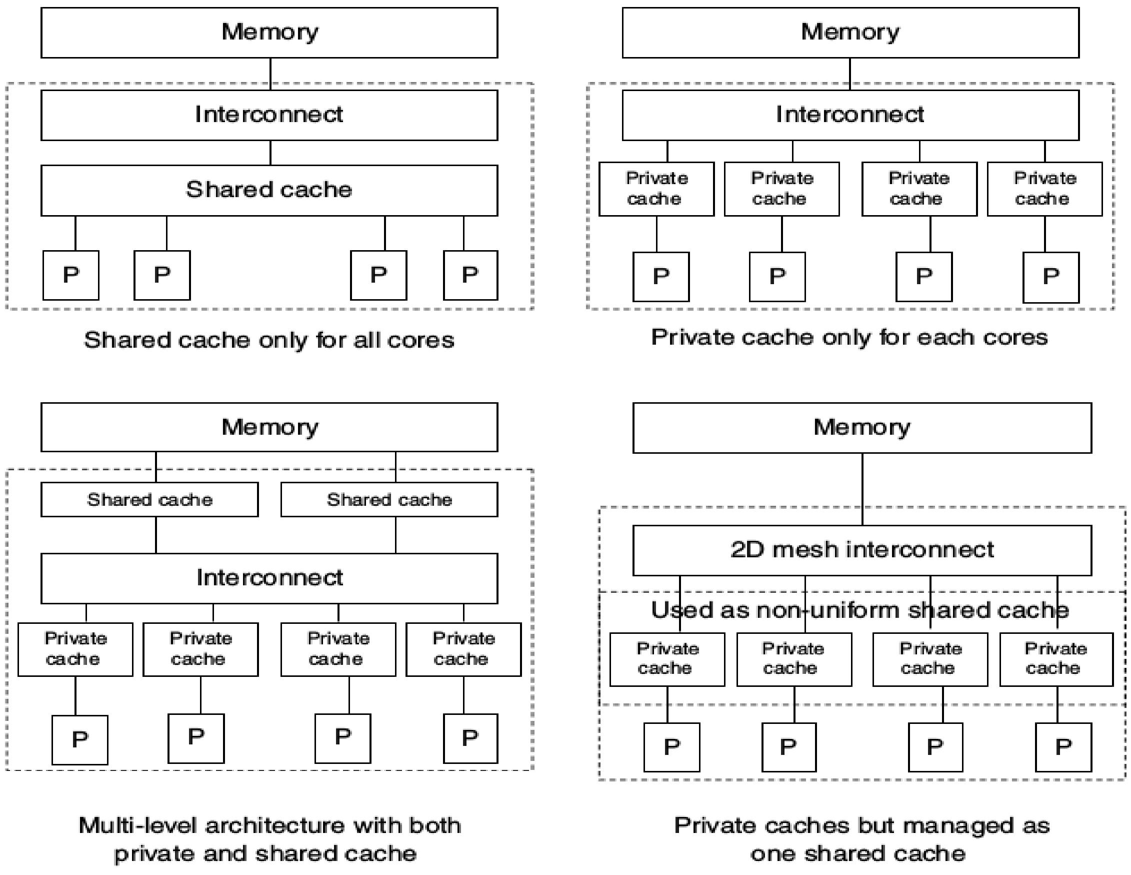
Memory
Interconnect
Shared cache
P
Shared cache only for all cores
Memory
Interconnect
Private cache
P
Private cache only for each cores
Memory
Shared cache
Shared cache
Interconnect
Private cache
P
Multi-level architecture with both private and shared cache
Memory
2D mesh interconnect
Used as non-uniform shared cache
Private cache
P
Private caches but managed as one shared cache

Figure 1.4: Cache Memory Architectures [33]

cores and robust synchronization mechanisms need to be implemented. However, it is very difficult to achieve hardware-only synchronization mechanisms and the more efficient solutions rely mostly on the software with some architectural support.
Most of the synchronization implementations do not scale well as the number of core increases,as the synchronization overhead becomes significant. In order to implement mutexes, the cores access shared memory and this requires some inter-core communication. As the number of cores increases, the overhead introduced by the shared memory accesses becomes significant. A distributed lock algorithm that scales almost linearly both in terms of system performance and in hardware cost is presented in [29]. The implementation uses local memories and message passing and is only bounded by memory bandwidth.
Additionally, as the number of cores increases, the cost of maintaining a coherent memory increases as well, as delay and latency incur by accessing frequently modified memory areas.
Memory bandwidth is also going to become a major concern as the amount of data required by the cores rises. The total amount of data needed is equivalent to the necessary memory bandwidth. If the number of transistors is going to increase with the existing pace, memory bandwidth will require the same trend in order to cope with the demand. However, this has not yet been achieved (Figure 1.5).

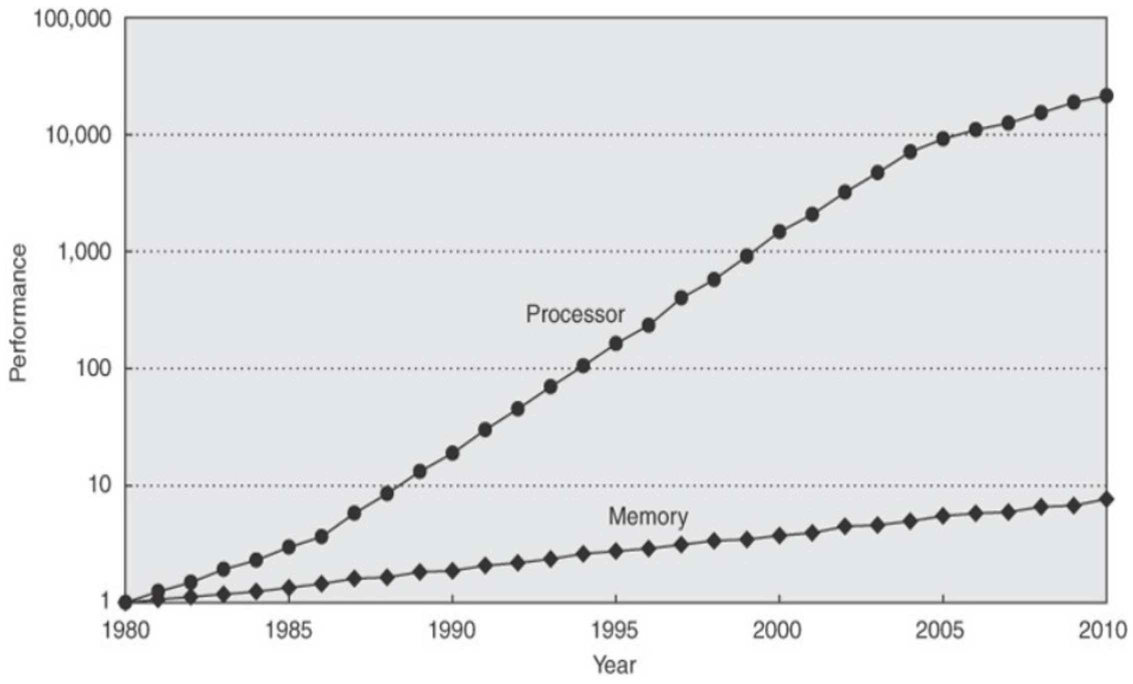

Figure 1.5: Processor and memory performance gap [23]

1.2.5 Future Trends

As multi-core processors evolve and the needs for the number of cores increase, issues regarding power consumption, synchronization and memory arise and require attention. Therefore, some innovative approaches have been given as possible solutions to the problems that cause bottlenecks to the scalability of many-core systems [33].
One big challenge of many-core systems is to find alternatives for the on-chip interconnect. As the number of cores is on the rise, the interconnect becomes the bottleneck, as it is limiting the system's throughput and is responsible for most of the system's power consumption. Regarding the types of on-chip interconnect illustrated in Figure 1.4, the Mesh interconnect performs better, as the wiring is shorter.
A concept that may introduce improve the on-chip interconnect in the areas of power consumption, latency and delay is 3D stacking. In the case of a 2D 32$\times$ 32 mesh interconnect of 1024 cores, the number of hops reaching across the chip would be 62, making 31 steps both horizontally and vertically. However, if the cores were placed in a 4 layer stack of 256 16$\times$ 16 mesh, then this number would reduce to 33, as the required cost would be 3 across the levels and 30 inside the same level. Similarly, memory can be packaged on top the processor cores, enabling higher speed interconnects.

Another approach is via the optical on-chip networks. These can offer lower power consumption and occupy less silicon area. The basic underlying principle is that a single optical waveguide can carry multiple signals on different wavelengths simultaneously, while having minimal loss across the transmission line. In order to reduce the synchronization overhead, the concept of transactional memory has been introduced, however it has not

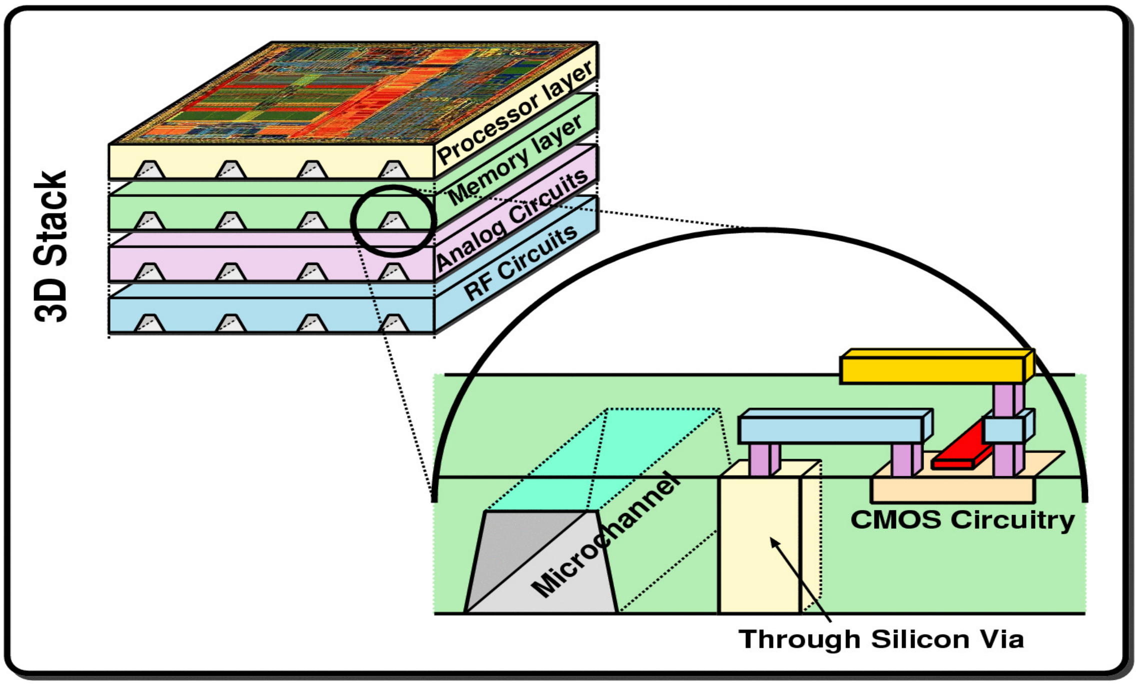

Figure 1.6: Scheme of 3D-IC stack and vertical interconnection [16]

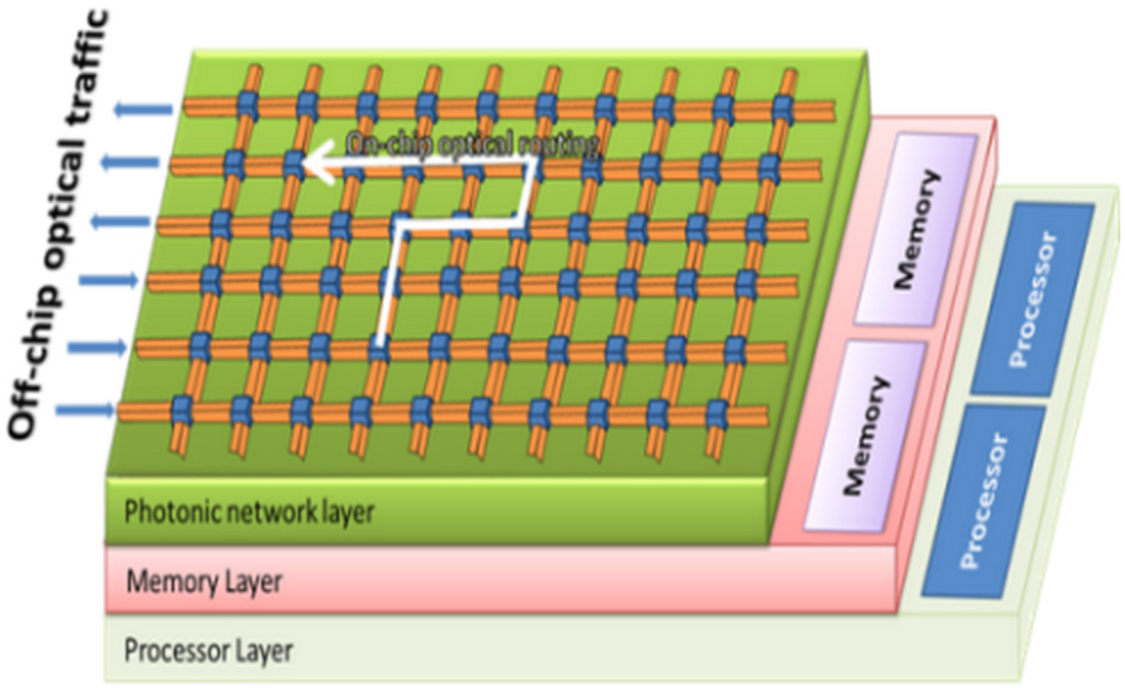

Figure 1.7: 3D silicon processor chip featuring on-chip nanophotonic network [19]

beed adopted by the industry yet. Transactional memory enables a thread to progress by performing its own updates until its own transactions are over. These changes are visible only to itself, but after completion the memory location is checked for concurrent accesses. If only an individual access has been performed, all changes are atomically commited and become global. Otherwise, the thread's transactions are rolled back and re-executed. Transaction memory enables concurrent updating of shared memory locations but only works well in cases where the probability of contention is very low.

1.2.6 Examples of Many-Core Systems

Intel Xeon Phi Coprocessor

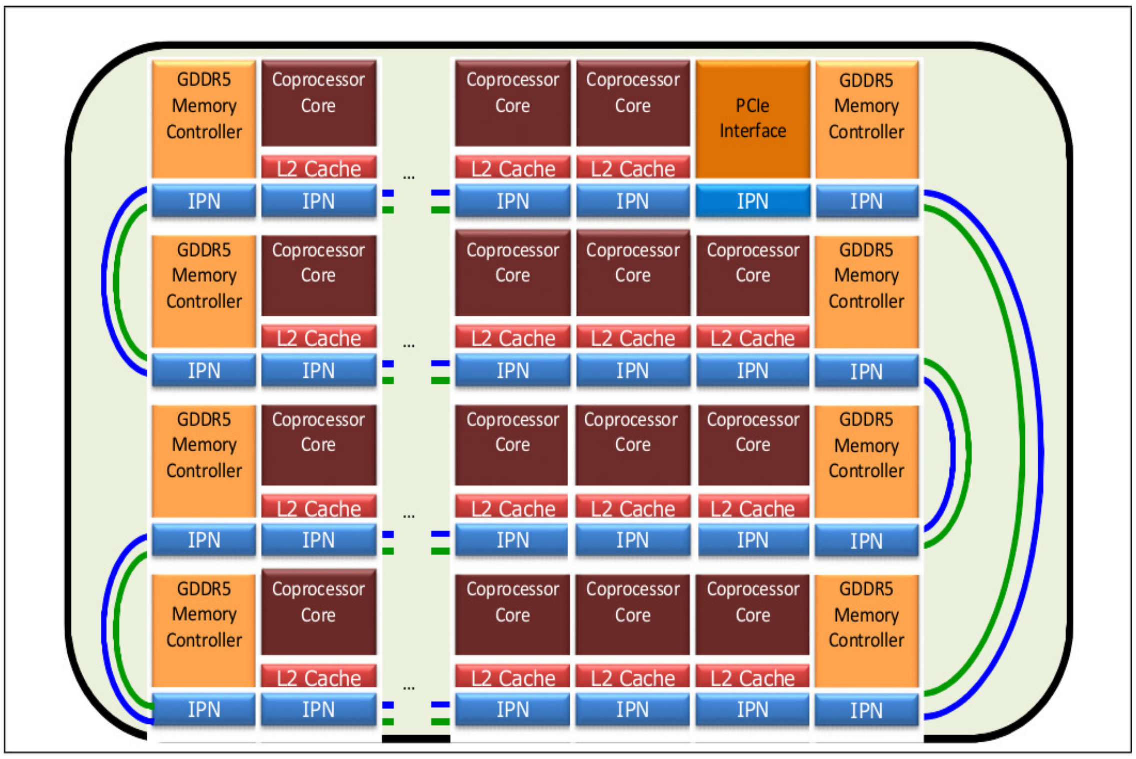

Figure 1.8: The Manycore Integrated Core (MIC) architecture on Xeon Phi [11]

Intel announced in 2012 a new processor product family named Intel Xeon Phi. This new processor features the Many Integrated Core(MIC) architecture. The cores of this architecture are Pentium based (version P54C) and use the x86 instruction set. The MIC architecture targets applications that require highly parallel processing in order to utilize a high degree of parallelism in its simple Pentium based processor cores.
In Figure 1.8 the general structure of the Xeon Phi coprocessor architecture is displayed. The cores, PCIe Interface logic and memory controllers are connected via a ring Interconnect, the Interprocessor Network (IPN) ring.
Regarding L2 caches, these are viewed as distributed across the cores, but can also be thought as a fully coherent cache with a total size equal to the sum of those attached to the cores. Data can be copied to individual cores to provide the fastest possible local access, but can also be found as a single copy for all cores to provide maximum cache capacity.
The Xeon Phi coprocessor can support up to 61 cores with a total of 30.5 MB L2 cache

and 8 memory controllers with 2 GDDR5 channels each.

The core architecture of the processor is displayed in Figure 1.9. The scalar unit is

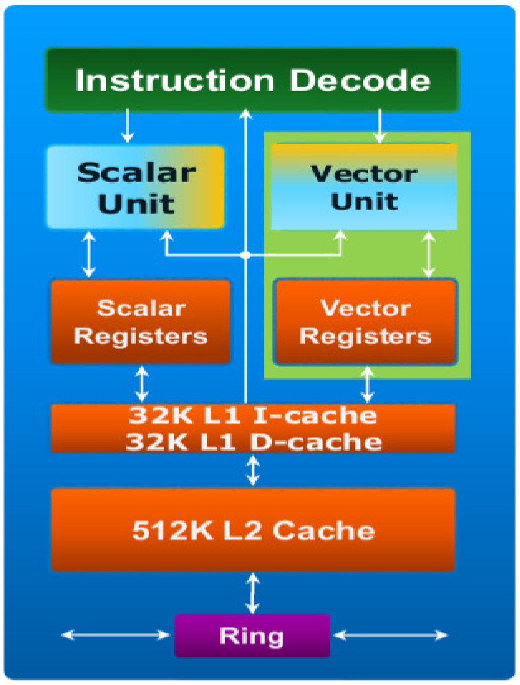

Figure 1.9: The Xeon Phi Coprocessor core [6]

based on the Pentium architecture, but Xeon Phi also has a vector unit implemented. L1 and L2 caches are fully-coherent. Another interesting feature is node Power and Thermal Management, which includes power capping support.

Intel SCC platform

The Intel SCC platform is a x86 48-core processor which provides a run-time and programming environment. In terms of architecture, the SCC platform features 24 tiles, each consisting of two blocks featuring a P54C core (second generation Intel Pentium processor), 16KB instruction and data L1 caches as well as a unified 256KB L2 cache. In addition, the platform features a A Mesh Interface Unit(MIU) allowing the mesh and the interface to operate at different frequencies, a 16KB Message Passing Buffer and two test-and-set registers. The architecture is illustrated in Figure 1.10.

Each tile also includes a router, which enables it to work with the Mesh Interface Unit (MIU) to integrate the tiles into a mesh. The MIU partitions data into packets to transmit them onto the mesh and puts together data received from the mesh using a round-robin scheme to arbitrate between the two cores on the tile. There is also a great number of off-chip memory which ranges from 16 to 64 GB of DDR3 RAM, controller by four memory controllers. Additionally, a router is connected to an off-package FPGA to translate the mesh protocol into the PCI express protocol, allowing the chip to interact with a PC serving as a management console.

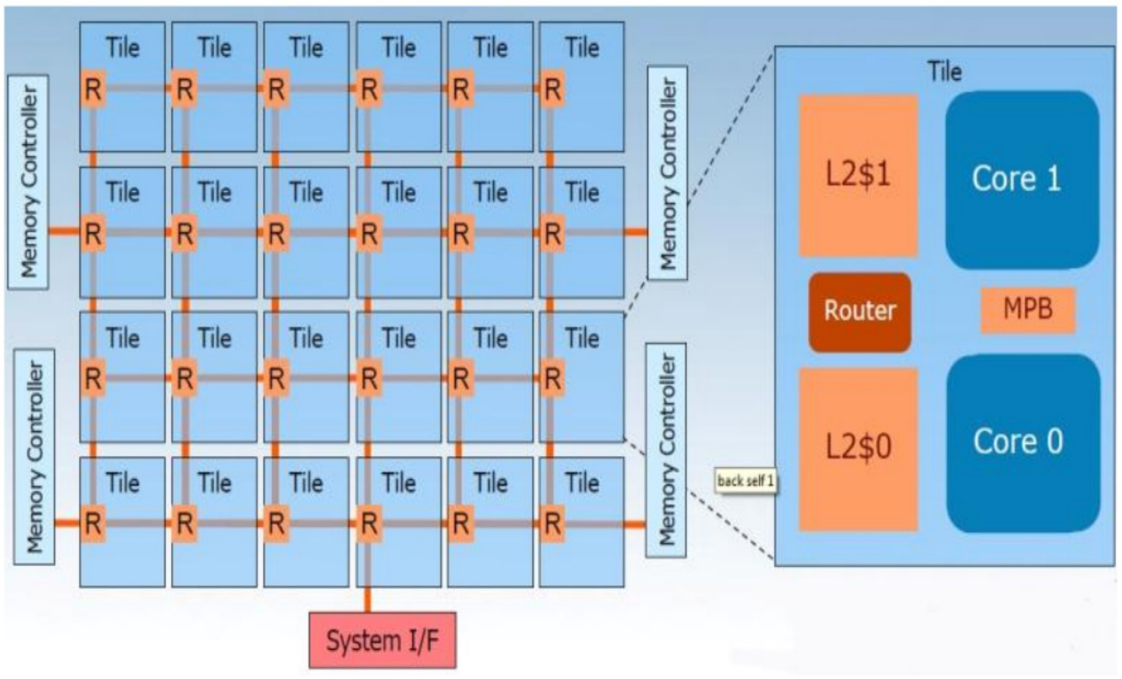

Figure 1.10: Scematic of the Intel SCC platform architecture [30]

On each tile, a message passing buffer (MPB) is included, which provides a fast, on-die shared SRAM, as opposed to the bulk memory accessed through four DDR3 channels. This message passing buffer is 16KB in each tile. Its memory is shared among all the cores on the chip. With 24 tiles, the SCC provides 384KB of message passing buffer. When a message-passing program sends a message from one core to another, internally it is passing the data through the message passing buffers on the chip [27],[30].

Tilera Tile Architecture

Tilera has introduced a multicore architecture that is based on the MIPS ISA and can currently scale up to 72 cores. The Tile Architecture is based on building a two-dimensional array of processor elements (given the name tiles) with a two-dimensional mesh network interconnect. This scalable architecture manages to provide low latency inter-tile communication and high bandwidth. All the tiles in the array are identical and include on-chip routers for data communication. As the tiles communicate with each other via point-to-point connections inside the Mesh interconnect, the tile array is energy efficient and offers significant scalability.
Each tile is a computing system that can support an entire Operating System. The Tile includes a 32-bit integer Processor Engine, a Cache Engine and a Switch Engine, which is the controller that coordinates all components of the tile into a powerful computing element. The Processor engine is a three-way VLIW processor that is capable of executing up to three operations per cycle and provides protection, support for memory management and Operating Systems. Moreover, the Cache Engine includes the TLBs, caches and

cache sequencers of the tile, as well as a DMA engine that coordinates data communication among tiles and between tiles and the external memory. Finally, the Switch Engine implements six different networks. Dynamic networks on the Processor facilitate the routing of packet-based data among tiles, tile caches, external memory and I/O contollers.
Figure 1.11 illustrates the TILEPro64 Processor organized in an eight by eight tile array [12].

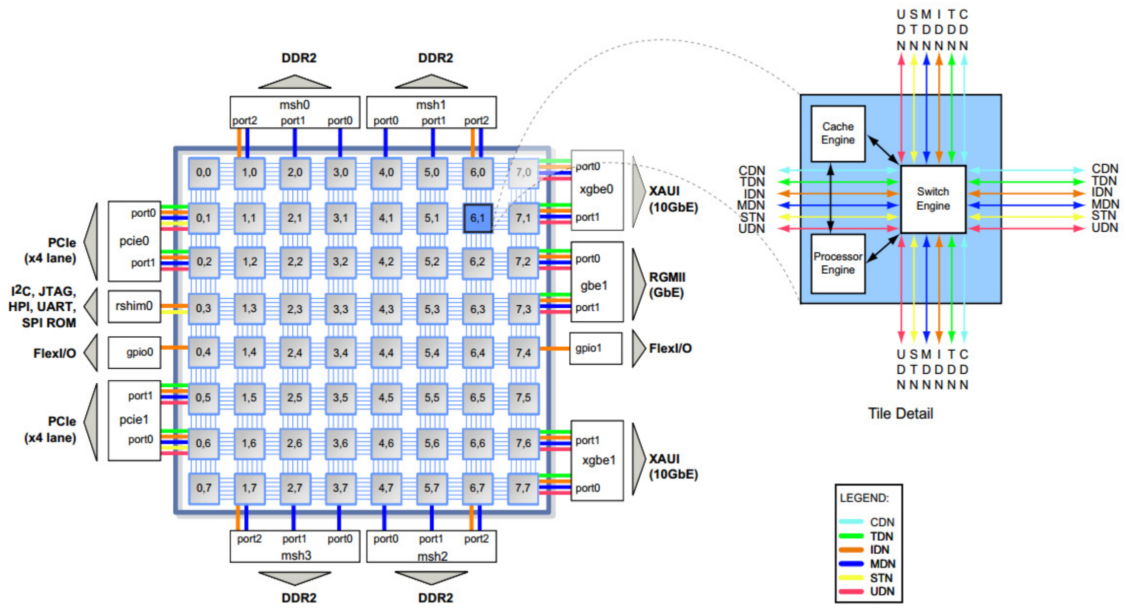

Figure 1.11: Scematic of the Tile Processor Architecture [12]

1.3 Resource Management

Traditionally, processors consisted of one cores or a small set of cores. For these types of systems, the problem of scheduling the tasks was responsible for resource allocation. However, as we move towards Many-Core processors resource allocation has become more complex and involves taking the core position into account. This process that incorporates the spatial aspect of the cores, is called mapping.

1.3.1 Scheduling

The history behind the evolution of schedulers as well as the state of the art is presented in [36]. Initially, schedulers were responsible for the execution time distribution of the processor among threads. Over time, the schedulers became responsible of establishing fairness, showing respect to priority, enabling real-time applications and of ensuring that all threads make progress. However, schedulers for single-core processors have reached a point, where there was not enough room for optimizations.
As processors evolved towards having multiple cores, schedulers needed to space share. In

other words, the Operating System scheduler needed to decide, to which core each thread -given an execution time interval- should be assigned. In multi-core processors, cores share resources that include caches and memory controllers. As a result, applications can display significant decrease in performance if they share resources with processes on neighboring cores.

To compensate for this performance loss, contention-aware schedulers have been introduced. This type of scheduler identify threads that share a lot of resources and threads that share minimal resources. In the sequence, it manages to schedule the threads with respect to minimizing the loss in performance due to shared resource contention.

Whereas threads from different applications might experience performance loss due to resource-sharing, threads of the same application might benefit for the same reason. Data-sharing threads might be more productive if the share the last level cache. Improved performance is realized also by sharing the same prefetching mechanisms.

As processors grow in number and the on-chip resources become abundant, scheduling also needs to evolve in order to cope with the changes in hardware. Traditional schedulers have managed individual threads. However, the concept of application containers has been introduced. Following the need for increased parallelism, applications will be running on top of runtimes that will have their own thread or task schedulers. In this way, a modular and more scalable approach will be realized, one similar to the parallel programming frameworks use already by having their own thread scheduling.

A similar approach to the management of application containers has been used in hypervisors, where virtual machines as regarded as individual entities. Memory and CPU are assigned to them in a fixed manner and the virtual machine itself is responsible for the resource management. An example of resource management for application containers is the Tesselation Manycore OS [9], in which the Cell Model is proposed as the basis for scheduling. A Cell is a container for parallel applications that provides guaranteed access to resources, a behavior similar to the one hypervisors provide for their virtual machines discussed previously. Tesselation allocates resources to containers as space-time quantities, such as " 4 processors for 10% of the time " or " 2 GB/sec of bandwidth ". Although Cells may be time-multiplexed, hardware thread contexts and resources are gang-scheduled such that Cells are unaware of this multiplexing. Resources allocated to a Cell are exclusively managed by that Cell until Tesselation requests the resources to be revoked. Once the Cell is mapped, a user-level scheduler is responsible for resource management within the Cell. Finally, Cells can communicate with each other via special channels.

1.3.2 The mapping problem

An overview of the mapping problem as well as a taxonomy of the different mapping classes is presented in [30]. The concept of Many-core SoC introduces new challenges in allocating resources amongst the cores. To date, resource allocation was limited in scheduling the tasks in one core or a very small set of cores. In Many-core systems, re-

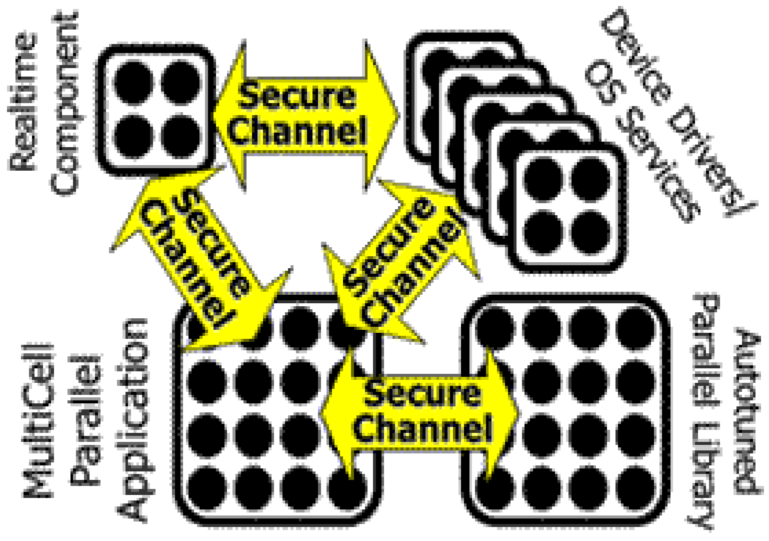
Realtime Component
Secure Channel
Device Drivers/ OS Services
Secure Channel
Secure Channel
MultiCell Parallel Application
Secure Channel
Autotuned Parallel Library

Figure 1.12: Inter-cell communication [9]

source allocation has a spatial aspect as well. An application comprises of tasks being executed in parallel and every task has to be assigned to a core. The choice of core has to take into account its position as well. This is the reason why we refer to this process as mapping. The idea of resource mapping is illustrated in Figure 1.13.

Calculating a cost efficient mapping for a given application in a short amount of time

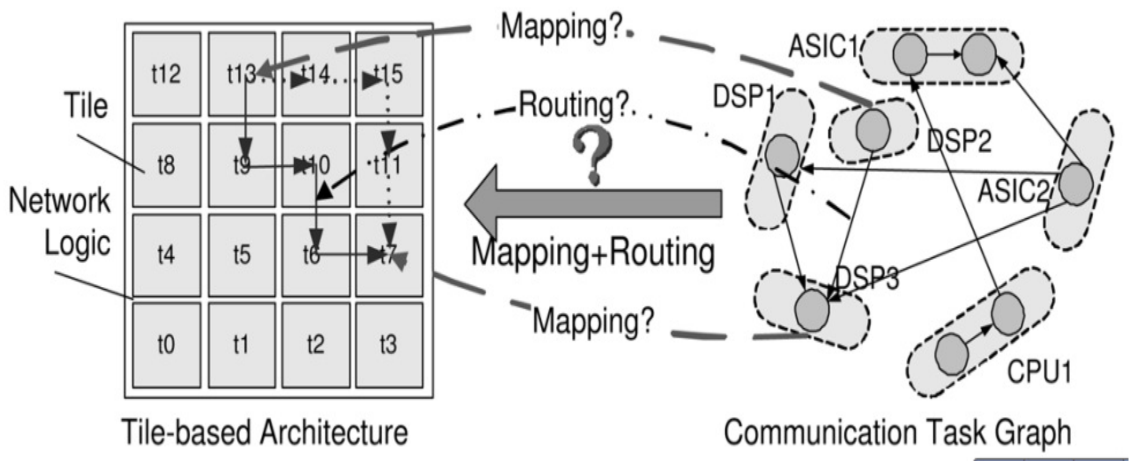

Figure 1.13: The mapping/routing problem [20]

is crucial. Some common goals are to maximize performance, minimize distance between tasks or minimize energy consumption in case of heterogeneous platform. Since the mapping process is an intermediate step between the request for an application to start and the actual initialization of the application, the process must be as fast as possible. As always, there is a variety of design decisions to be made as far as the characteristics of the mapping are concerned. The most important are presented in the following sections.

Design-time vs Run-time mapping

In the case of Design-time mapping, the nodes an application can run on are designated before run-time. This designation cannot be altered during run-time. Apparently, this technique can be employed only when small amount of application can be run on such a processor and only for specific scenarios, which is not very common in modern systems. The alternative of design-time mapping is the run-time one, in which resources are allocated dynamically as applications enter and exit the system, providing the necessary flexibility for the platform to function properly. An interesting fact is that run-time mapping is the only way to tackle the unpredictability of the incoming applications. Another characteristic is that run-time mapping manages to adapt to the available resources. Those vary over time, due to applications running simultaneously. Run-time information can be incorporated to further reduce the cost of running an application. In addition, it enables unforeseeable upgrades after first product release time, e.g. new application and new or changing standards as well as to avoid defective parts of a SoC. Larger chips mean lower yield. The yield can be improved when the mapping algorithm is able to avoid faulty parts of the chip. Also, aging can lead to faulty parts that are unforeseeable at design-time. Finally, it can be used with reconfigurable hardware where the type of available processing elements can vary over time.
A downside of run-time mapping is the extra overhead it adds to the execution of an application, since it is executed between the request from the OS and the actual execution of the tasks. Hence, it is crucial that the mapping is calculated fast, so that it doesn't overburden the system.

Centralized vs Distributed mapping

Apart from defining the moment (run-time or design-time) the mapping occurs one must also define on which tiles the mapping algorithm will be executed. In this case the available design choices are to perform the mapping calculations either on a distributed or centralized manner [20]. Centralized mapping utilizes one or a small set of cores, Centralized Managers (CM), to perform the mapping for every application that arrives. These cores are burdened with the responsibility to calculate the mapping for the whole system. This mapping scheme can cause some problems [17].
Monitoring traffic is increased in volume, as the Centralized Manager needs to collect data from the whole chip during the mapping -performed at run-time-, which causes traffic on the wires, possibly stalling the execution of already running tasks. In addition, the computational cost to calculate the mapping for the whole chip at once is high. Another drawback is that it is prone to failure. If the Centralized Manager fails for some reason, the mapping can't be performed at all. Moreover, the Centralized Manager becomes a point of hot-spot as every tile sends the status of the Processing element to it. This increases the chance of bottleneck issues around the manager. Consequently, because of all these reasons scalability issues arise. As Many-core chip grow in size and more Processing

Elements will be added, the computational effort of mapping and the traffic it will create will increase exponentially, thus rendering the computation very expensive and the scheme ineffective.
On the distributed mapping scheme, the effort of the computation is distributed, as the name suggests, on several tiles across the chip, Local Managers (LM) as presented in [32], and they may even change from one mapping to the next. In this way, the problems of the Centralized mapping are solved.
First, monitoring traffic is decreased in volume, as the Processing Elements only need to send the data to their closest Local Manager, and this way they travel less on the chip. When mapping decisions have to be made involving different regions, the communication is limited only between Local Managers. In addition, Local Managers only need to perform the mapping computation for the area of the chip they are responsible for, or for some designated tiles. This way, the computation demanding problem is divided in less demanding ones. Apart from these, there are no issues of single point of failure or hot-spots, since the smaller portions of the computation can be performed on any tile. Finally, it scales very well with larger number of cores, since all that is needed is some more light-weight Local Managers, whose individual computation effort isn't increased.
With distributed mapping, two major downsides occur. First, there is the need for synchronization between the managers, so as to achieve a well-balanced and fair mapping. Secondly, it is reasonable that a centralized approach can provide a better mapping considering the fact that it has an overall view of what is going on in the platform. However, we can achieve results comparable to the optimum mapping with the distributed approach and taking into account its aforementioned advantages, distributed resource management seems to be the only option [4]. The following Figure (Figure 1.14) illustrates the principles of centralized and distributed mapping.

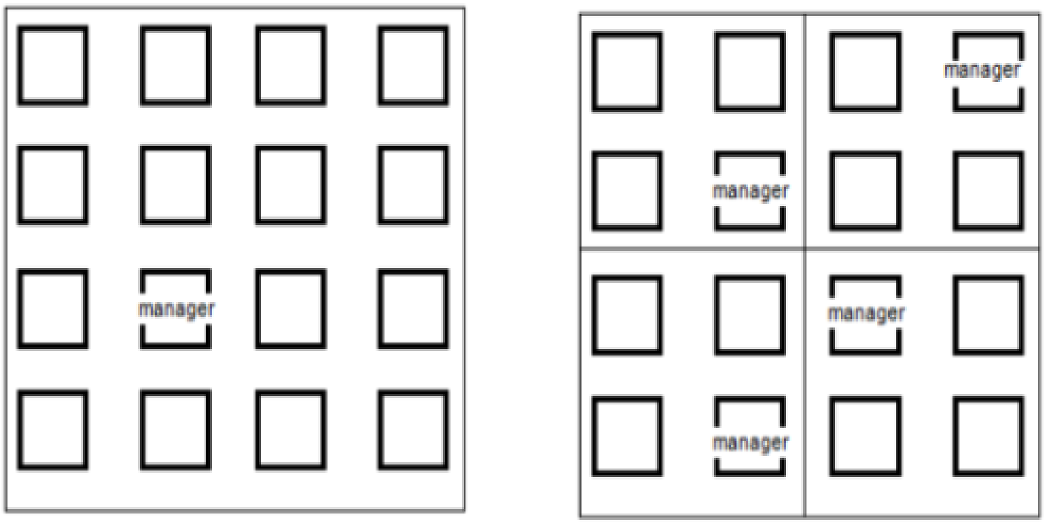

Figure 1.14: Processor tiles examples using centralized and distributed mapping [30]

1.4 Economic principles

One of the fundamental models used in economics is the supply and demand model for a competitive market. A competitive market is one that has many buyers and sellers, none of which has the ability to individually influence the price of the market.
In addition, a market is defined in terms of a specific group of individuals who would potentially trade with each other. Markets can both have very broad or very narrow scope (e.g. internationally vs locally in a single community).
The basic components of the model (demand and supply curves, the market equilibrium and the factors that cause these to change) are presented below [18].

Demand

The definition for Demand is the following:
An individual's quantity demanded for a good specifies the amount an individual would choose to buy of a good over some time period given a particular price that must be paid for the good and all other constraints on the household (e.g., income).
The demanded quantity reflects a choice individuals make dependent on many factors and is also hypothetical and therefore independent of the availability of the good at a certain price. Additionally, the quantity demanded causes the price to stress.
The law that drives the behavior of the demand mentions that when the price of a good rises, and everything else remains the same, the quantity of the good demanded will fall. This law indicates that the demand curve has negative slope.
A graphical representation of the demand curve can be found on Figure 1.15. An approximation of that curve can be represented with a line with negative slope and positive offset.
There are several factors that can affect the demand. On the demand curve we can see a point (P_1,Q_1). Factors that can shift that point on the demand curve are changes in wealth, in the number of consumers, in the prices of the good or related goods, in customer tastes as well as in customer (price) expectations. For example, if an increase in wealth is observed, an increase in the quantity demanded of the good at any given price would be expected.

1.4.1 Supply

The definition for Supply is the following:
An individual's quantity supplied of a good is the specific amount its managers would choose to sell over some time period, given a particular price for the good and all other constraints on the firm.
The supplied quantity reflects a choice firms make regarding the quantity they would be willing to sell at a given price. The supplied quantity also hypothetical and therefore independent of the customers' willingness to pay for the supplied quantity a the given price.
Similar to the demanded quantity, the supplied quantity causes the price to stress.

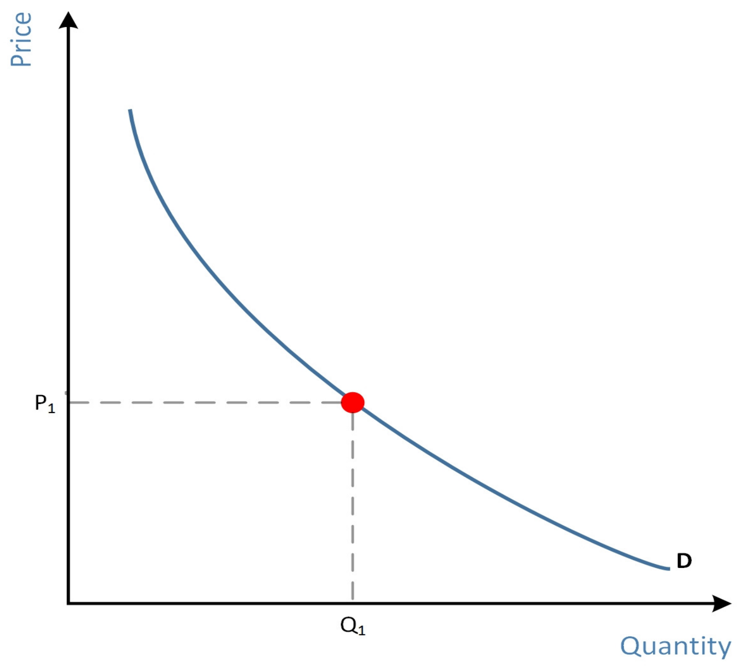

Figure 1.15: Demand Curve

The law that drives the behavior of the supply mentions that when the price of a good rises, and everything else remains the same, the quantity of the good supplied will increase. This law indicates that the supply curve has positive slope.
A graphical representation of the supply curve can be found on Figure 1.16. An approximation of that curve can be represented with a line with positive slope and positive offset.

There are several factors that can affect the supply. On the supply curve we can see a point (P_1,Q_1). Factors that can shift that point on the supply curve are changes in the number of producers, changes in the prices of the good or related goods, changes in technology as well as changes in (price) expectations. For example, if an increase in the number of producers is observed, this will lead to an increase in supply. Then, the supply curve shifts rightwards.

1.4.2 Market equilibrium

One principle that is governing markets is that they tend to move towards an equilibrium. Changes can occur, but eventually the effects of the change will become partially compensated and the market will converge towards an equilibrium.
Competitive markets are in equilibrium when price has moved to a point where the quantity of the good demanded is equal to the quantity of the good supplied. This price at which we have an equilibrium point is called market-clearing price or equilibrium price and the quantity equilibrium quantity.
In Figure 1.17 we can see an equilibrium graphically. The equilibrium price and quantities in the graph indicate where the market will naturally settle down to given the current sup-

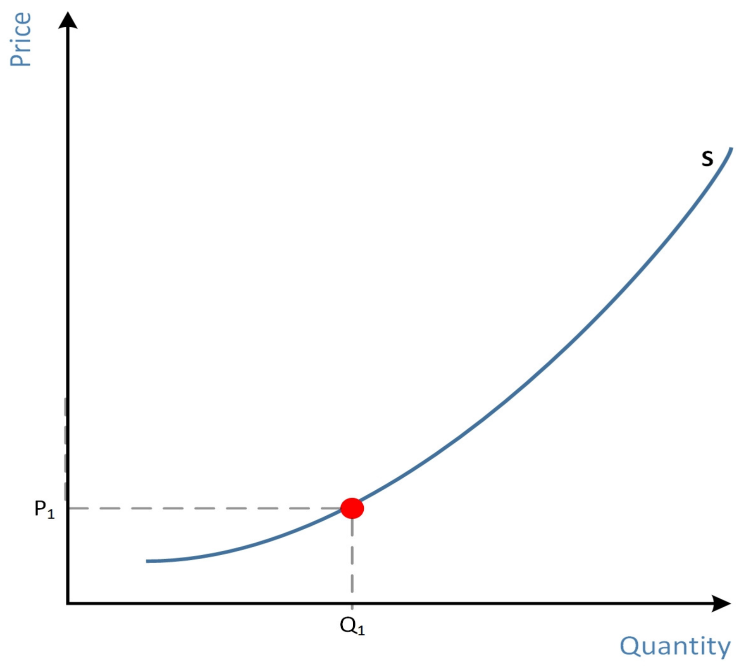

Figure 1.16: Supply Curve

ply and demand curves. If any of the quantities change, this can cause shifts in demand and shifts in supply. These changes have an impact on the equilibrium point, shifting it either across the demand or across the supply curve.

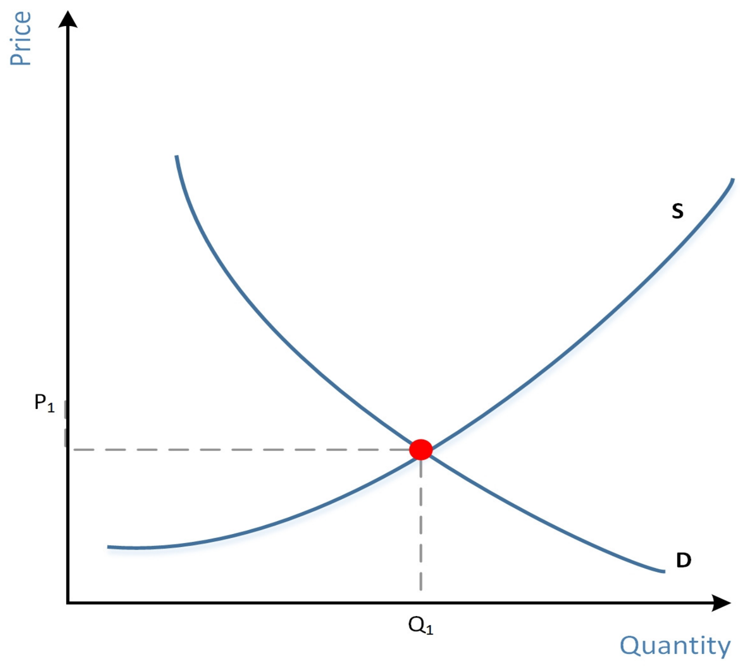

Figure 1.17: Market equilibrium

1.4.3 Auctions

An auction is defined as a process of buying and selling goods or services by offering them for bid. After bids are taken the item is sold to the highest bidder.
An overview of auctions and the underlying theory can be found in [26]. An auction format is a mechanism to allocate resources among a group of bidders. The auction model consists of the description of the potential bidders, the set of possible resource allocations (describing the number of goods of each type, whether the goods are divisible, and whether there are legal or other restrictions on how the goods may be allocated) as well as the values of various resource allocations to each participant. The mechanism receives all information in the form of bids, as potential buyers express their willingness to pay in that way. Additionally, the auction outcome, which includes the highest bidder and the amount of the bid (market clearing price) is determined on the basis of the received information. Another characteristic of auctions is anonymity, which indicates that the identity of the bidder does not affect the outcome of the auction, both in terms of the amount paid and who the winner is. Some common forms of auction are the following:

- The English Auction
  The English auction is the open ascending price auction. This type of auction is carried through by an auctioneer who begins with a low initial price and raises it, typically in small increments, as long as there are at least two interested bidders. The auction stops when there is only one interested bidder.

- The Dutch Auction
  The Dutch auction is the open descending price auction. The auctioneer begins at a high price so that no bidder would be interested in buying the object at that price. The price is progressively lowered until a bidder expresses interest.

- The Sealed-Bid First-Price Auction
  In this auction type, bidders conceal their bids and submit them. The person submitting the highest bid wins the object and pays what he bid.

- The Sealed-Bid Second-Price Auction
  Bidders conceal their bids and submit them. The person that submits the highest bid wins the object but pays the amount of the second highest bid.

At the time of the auction, the bidders have knowledge or no knowledge regarding the price of the object. If each bidder knows the value of the object to himself, then the auction is called one of private values. However, most often the worth of the object is unknown at the time of the auction to the bidder himself. This type is called of independent values, since each of the bidders can only make an independent and personal valuation of the object. To form this valuation, bidders sometimes use information, such as an expert's estimate to have an approximate of the true value and adjust their bid accordingly.

Chapter 2

Related Work

2.1 ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication [2]

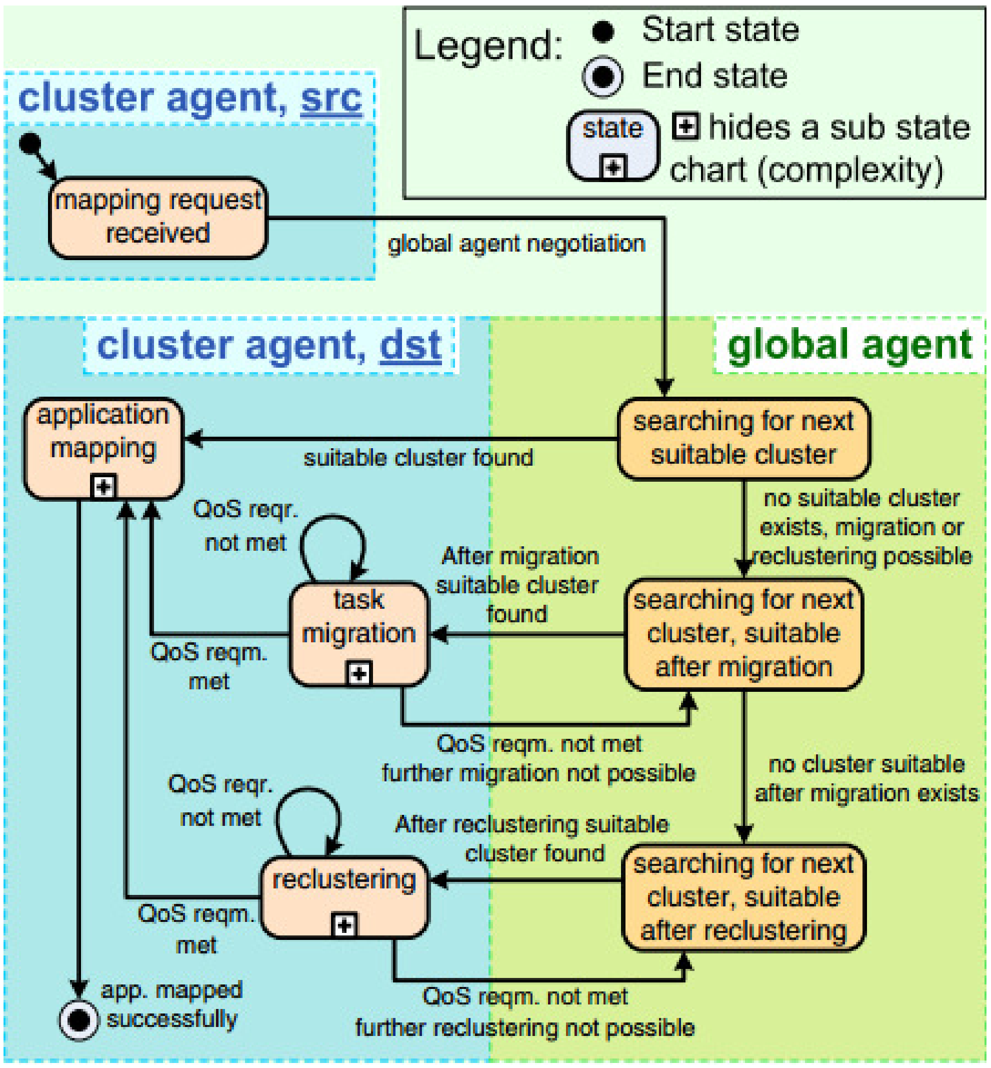
Legend:
● Start state
End state
state
hides a sub state chart (complexity)
cluster agent, src
mapping request received
global agent negotiation
cluster agent, dst
global agent
application mapping
searching for next suitable cluster
suitable cluster found
no suitable cluster exists, migration or reclustering possible
QoS reqr. not met
task migration
After migration suitable cluster found
searching for next cluster, suitable after migration
QoS reqm. met
QoS reqm. not met further migration not possible
no cluster suitable after migration exists
QoS reqr. not met
After reclustering suitable cluster found
reclustering
searching for next cluster, suitable after reclustering
QoS reqm. met
app. mapped successfully
QoS reqm. not met further reclustering not possible

Figure 2.1: Flow of the ADAM algorithm [2]

In this work, the authors present an agent-based distributed resource management algorithm for self-adaptive heterogeneous MPSoCs. The negotiation of cores takes place inside a virtual cluster. Such a cluster is an area of the platform that can be creates, resized and destroyed at run-time. Additionally, an agent is a computational entity acting on behalf of others. Its main characteristics are:

1. It is a small task close to the system.
2. It performs resource management.
3. It must be executable on any PE.
4. It must be migratable and recoverable.

5. It may be destroyed upon the destruction of the cluster.

A cluster agent is CA, is an agent responsible for mapping operations inside a cluster. A global agent GA is responsible for storing global information such as usage of communication and computation resources for each cluster and this information is used for selection and re- organization of the clusters. An overview of the mapping scheme is presented in Figure 2.1.

When a new mapping request is received from any tile, the Cluster Agent of the tile's

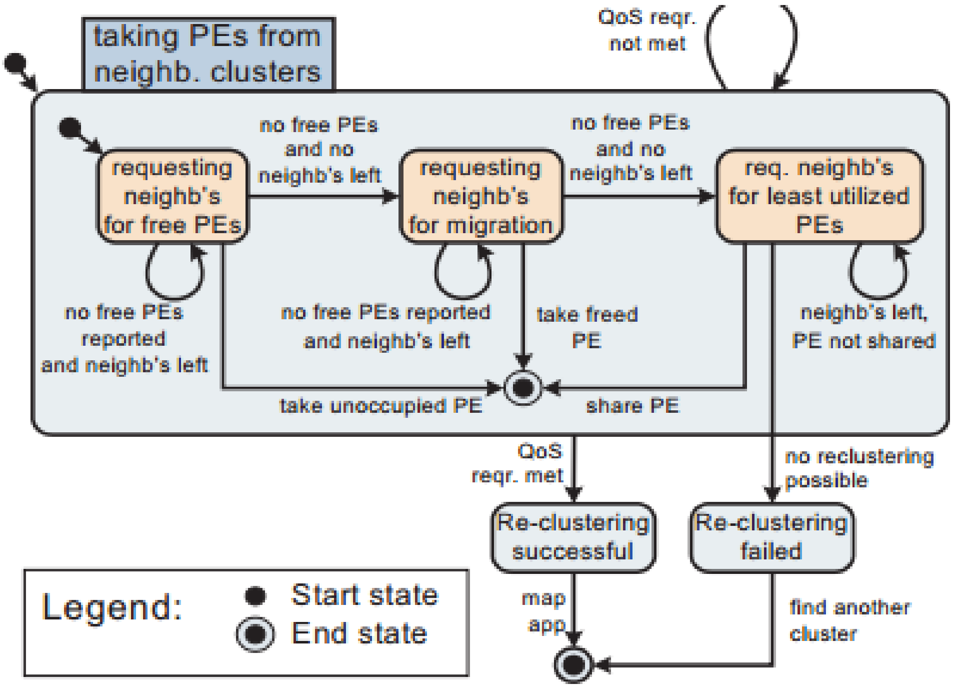
taking PEs from neighb. clusters
QoS reqr. not met
requesting neighb's for free PEs
no free PEs and no neighb's left
requesting neighb's for migration
no free PEs and no neighb's left
req. neighb's for least utilized PEs
no free PEs reported and neighb's left
no free PEs reported and neighb's left
take freed PE
neighb's left, PE not shared
take unoccupied PE
share PE
QoS reqr. met
no reclustering possible
Re-clustering successful
Re-clustering failed
map app
find another cluster
Legend:
Start state
End state

Figure 2.2: The re-clustering process of the ADAM algorithm. [2]

cluster communicates with the Global Agent, indicating the request. At this point, the Global Agent performs the Suitable Cluster Negotiation Algorithm, which finds a cluster capable to fit the whole application. The Suitable Cluster Negotiation Algorithm checks if there are enough free tiles in every PE type and resource requirement class for all the tasks in the application.

In case the algorithm fails to find a suitable cluster then a task migration is employed is the most suitable cluster. The point of this task migration is to rearrange the tasks within a cluster in a way that the mapping of the new application will be feasible in the cluster. Still there is a probability that this will fail to produce an appropriate mapping as well. Then the technique of re-clustering is employed. Clusters can be changed in shape, resulting to the probable creation and destruction of some them. This process can benefit running applications since resources can be redistributed in a better way. However, it is an elaborate process that takes into account many parameters of the applications such as the requested energy and memory of the application. The flow of the reclustering algorith is presented in Figure 2.2.

After a cluster that can host the application has been found, that cluster's agent is responsible to perform the Run-time Mapping algorithm to appoint every task to a tile to be executed on. The best tile for every task is calculated using a heuristics, checking the tile's position in the cluster (tiles near the center are preferred), the volume of communication on the tile before and after the mapping and the resource requirements for the task to run on any tile.

2.2 A Divide and Conquer based Distributed Run-time Mapping Methodology for Many-Core platforms [3]

In this paper a distributed run-time resource management framework is presented for both homogeneous and heterogeneous platforms. The main idea of the proposed framework is illustrated in Figure 2.3.

The mapping is carried out in a distributed manner. In order to achieve that, the platform

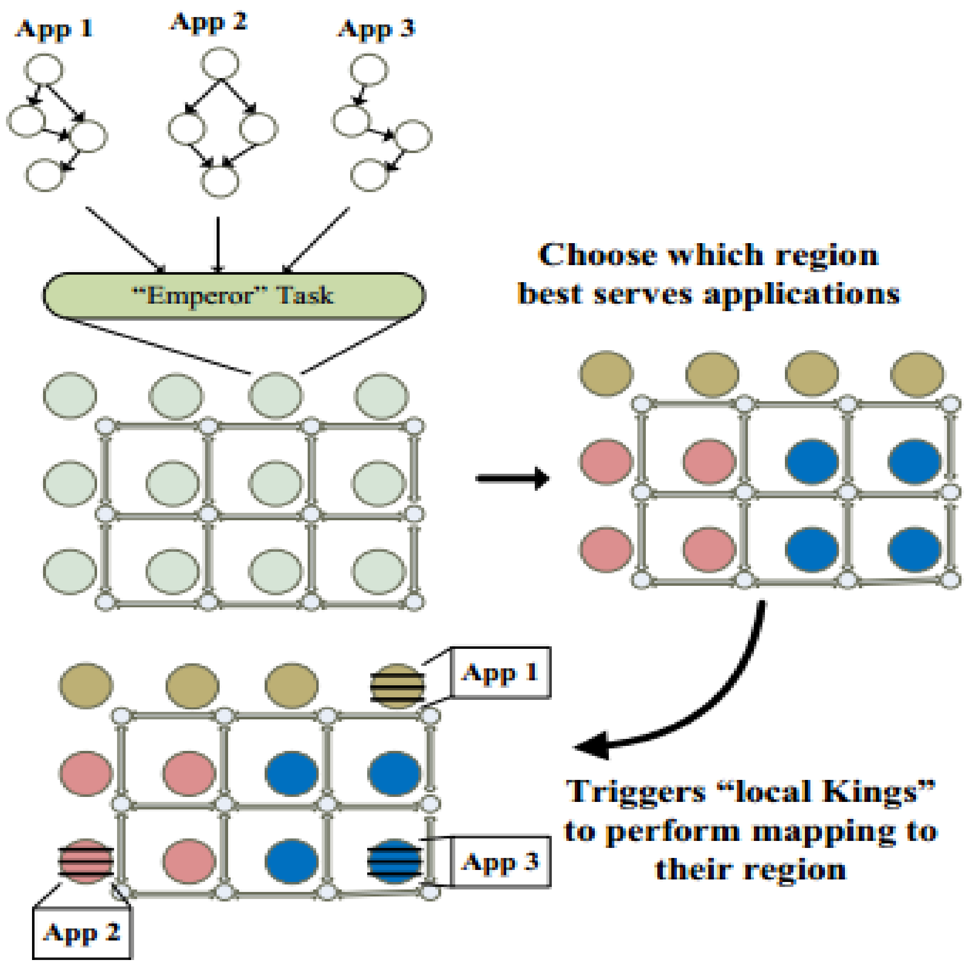
App 1
App 2
App 3
"Emperor" Task
Choose which region best serves applications
App 1
App 3
App 2
Triggers "local Kings" to perform mapping to their region

Figure 2.3: Main idea of the run-time manager of [3].

is partitioned in regions. A region is a subset of the set of all the tiles on the NoC with no fixed boundaries, and can be reshaped, created or abolished when necessary. The manner in which the partitioning is performed is different in homogeneous and heterogeneous platforms. The overall flow of the resource management algorithm for both homogeneous and heterogeneous platforms is presented in Figure 2.4.

Upon arrival of an application, a system-wide controller is invoked. This task is called

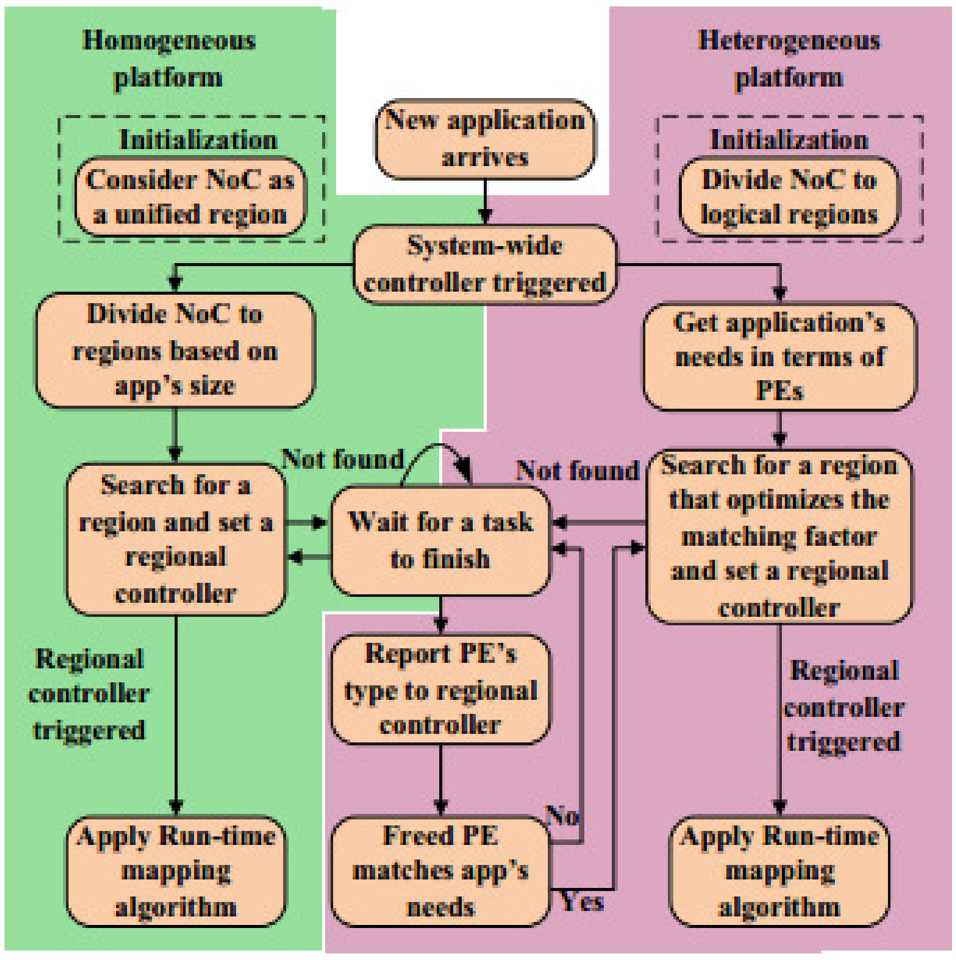
Homogeneous platform
Heterogeneous platform
New application arrives
Initialization
Consider NoC as a unified region
Initialization
Divide NoC to logical regions
System-wide controller triggered
Divide NoC to regions based on app's size
Get application's needs in terms of PEs
Not found
Not found
Search for a region and set a regional controller
Wait for a task to finish
Search for a region that optimizes the matching factor and set a regional controller
Regional controller triggered
Report PE's type to regional controller
Regional controller triggered
Apply Run-time mapping algorithm
Freed PE matches app's needs
No
Yes
Apply Run-time mapping algorithm

Figure 2.4: Flow of the run-time resource management of [3].

an "Emperor" task. It is a light-weight in terms of resource demands, it can be executed on any type of processing element of the platform and it is responsible for:

1. getting the requirements of the incoming application
2. selecting the appropriate region to map the application on
3. triggering other cores in the region to perform the run-time mapping algorithm. These regional controllers are referred to as "Local Kings".

In addition to the System-Wide Controller, in every region there is a specific node called a Regional Controller (RC). These tiles are responsible for any action involving the mapping in their respective region. Their code can also be executed on any processing element of the platform. A Regional Controller is responsible for:

- Computing the mapping for the region for which the controller is responsible for.
- Collecting data for the region.
- Communicating and exchanging data with the System-Wide Controller.

The System-Wide controller is responsible for selecting the region on which the application will be mapped on. After that it is responsibility of the Regional Controller to calculate the actual mapping and inform the System-Wide controller. As far as the mapping is concerned, the goal of the algorithm is to compute a fast and efficient mapping that minimizes the energy consumption from the execution of an application. After an initial mapping has been performed an iterative application node swapping process is employed in order to further reduce the total communication cost.
Comparing the RTM algorithm for heterogeneous and homogeneous NoCs its major difference is the way the regions are initialized. On homogeneous platforms during the initialization no regions exist, and are created for each new application, and abolished when the execution finishes its execution. On the other hand on heterogeneous platforms, the NoC is partitioned in regions from the very beginning, with the selection of number and size of the regions left on the designer's judgment and Regional Controllers are appointed on these regions right away. Regions can be reshaped or created to better accommodate new applications, but they can only be abolished if they are an empty set.
Finally, the concept of matching factor is introduced. This designer specified percentage value dictates how strict the mapping is as far as the preferred processing elements of an application are concerned. An MF of 100% dictates that the application can accept only its most preferred processing element, while an MF of 0% dictates that an application can accept any of the processing elements it is able to work on.

2.3 DistRM: Distributed Resource Management for On-Chip Many-Core Systems [25]

The authors of [25] present a distributed run-time resource management framework for malleable applications. As the name of the work suggests, the framework is called DistRM. The application model used for the examination of the proposed framework is the one proposed by Downey [15]. The framework is distributed in the sense that different cores, dispersed around the grid of cores, are responsible for performing mapping decisions and managing applications. These cores are called agents.
To manage regional information in a distributed way, the authors suggest that there is a directory service distributed to the entire platform. This directory service enables the agents to communicate with other agents without the need of broadcast communication. All available cores are split into evenly distributed clusters and each cluster contains one

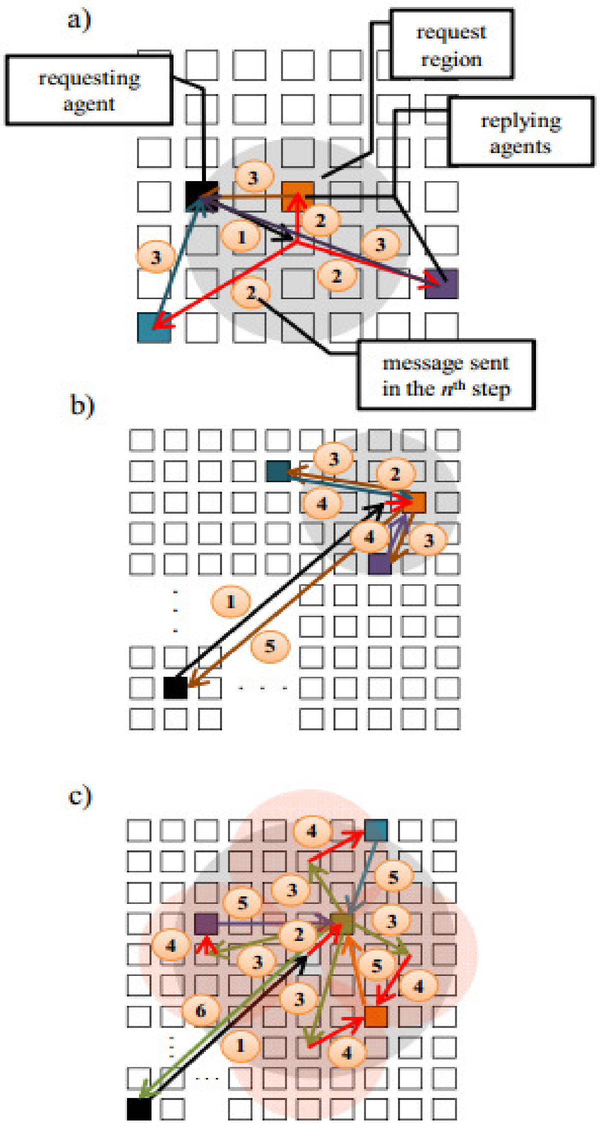

Figure 2.5: Communication between agents for a) local and b) distant target regions; c) a display the divide-and-conquer behavior for larger target regions [25].

directory service running on one of the cores. The agents register themselves at the directories corresponding to the cores occupied by the own application.
When a new application arrives on the system a randomly chosen core is triggered to initiate its agent. This is a means of load balancing without the need for global system state knowledge. This core acts as a seed for searching resources and there is a probability that this core will not be chosen for the application. The agent determines the cores the application will eventually run on. The agent searches in randomly chosen areas on the system which are gradually moving away from the core if no suitable core for the application is found. When such a request has to be made in a region further from a specific threshold, then this is consider a far request and the core in the center of the far region is delegated to discover a far request manager. It utilizes the knowledge of the

distributed directory service and forwards the request to the agent with most cores in the desired area. Figure 2.5 illustrates the interplay of the resource management agents and applications. The figure shows that an agent can communicate with all other agent for

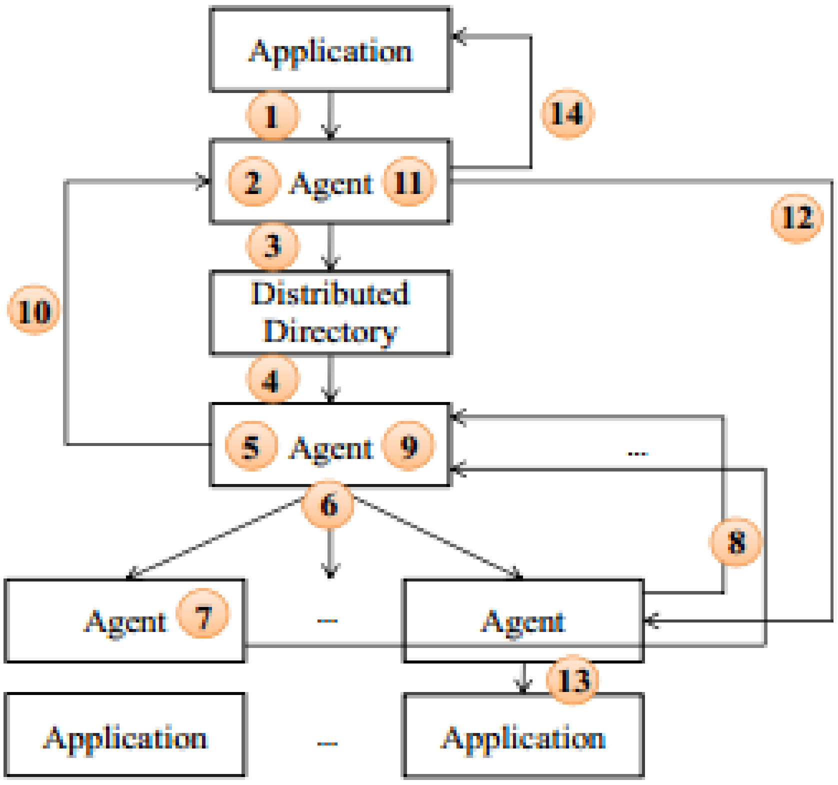

Figure 2.6: Interaction of components of the DistRM algorithm [25].

the bargaining of cores to take place. A core is offered to the requesting agent if its gain in speedup is greater than the loss in speedup of the agent offering it. To avoid constant trades of cores between two agents, a core trade takes place only if the gain of the receiver is significantly bigger compared to the loss of the offering agent. In any case of a core offer, the offering agent has to receive a reply dictating if its offer is accepted or not. The agents also communicate with the cores holding the regional information and with the cores of their application in order to reconfigure and resize it.

The unallocated cores of the platform are handled by a special category of agents called the idle ones. These agents handle cores when the platform is initialized or when an application has finished its execution and there is no need for its agent to exist anymore. They are located at fixed positions on the platform and form a regular grid distributed all over the system. When another agent asks an idle agent for a core, the latter always offers one provided that it handles at least one.

Finally the concept of self-optimization of an application is introduced. After an agent has discovered an initial set of cores for its application, it requests more cores in order to increase the speedup of the application. At some point at run-time, an agent sends request for cores to its nearby regions. This idea helps to optimize the mapping of application over time. To ensure that requesting additional resources does not happen too often, the delay between two self-optimization processes is doubled after the completion of one.

2.4 A taxonomy of market-based resource management systems for utility-driven cluster computing [35]

The aim of this paper is to develop a taxonomy that characterizes and classifies how market-based Resource Management Systems (RMS) can support utility-driven cluster computing in practice. A categorization of market-based RMSs is illustrated in Figure 2.7.

In utility-driven cluster computing, consumers have different requirements and needs

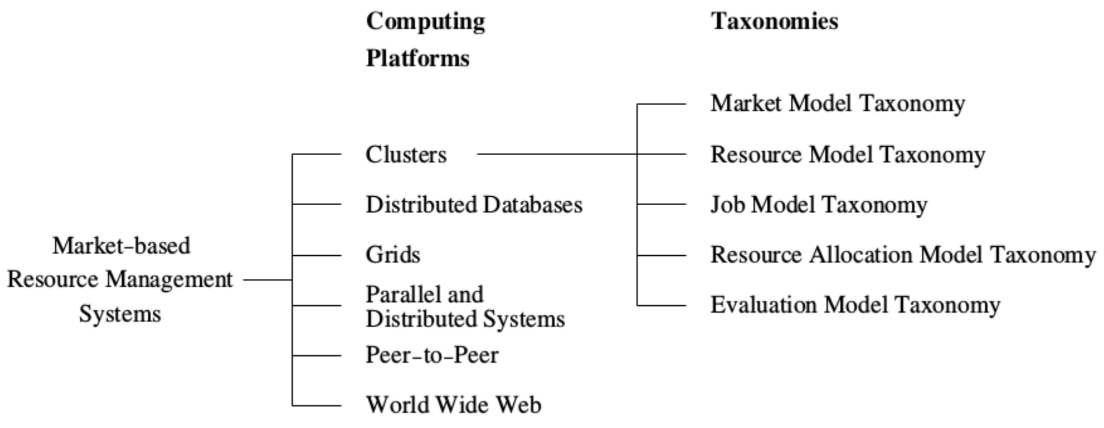

Figure 2.7: Classification of market-based RMSs [35].

for various jobs and thus specify their requirements and preferences for each respective job using Quality of Service (QoS) parameters during job submission to the cluster RMS. Market-based cluster RMSs need to support three requirements in order to enable utility-driven cluster computing:

1. Provide users with a means of generating their QoS needs and valuations.
2. Employ policies to create a translation of the valuations into resource allocations.
3. Have mechanisms that enforce the resource allocations in order to achieve each individual user's perceived value or utility.

In Figure 2.8 an abstract model for the market-based cluster RMS is displayed. Its purpose is to identify generic components that are essential in a practical market-based cluster RMS and outline the interactions between these components. Thus, the abstract model can be used to study how existing cluster RMS architectures can be leveraged and extended to incorporate market-based mechanisms to support utility-driven cluster computing in

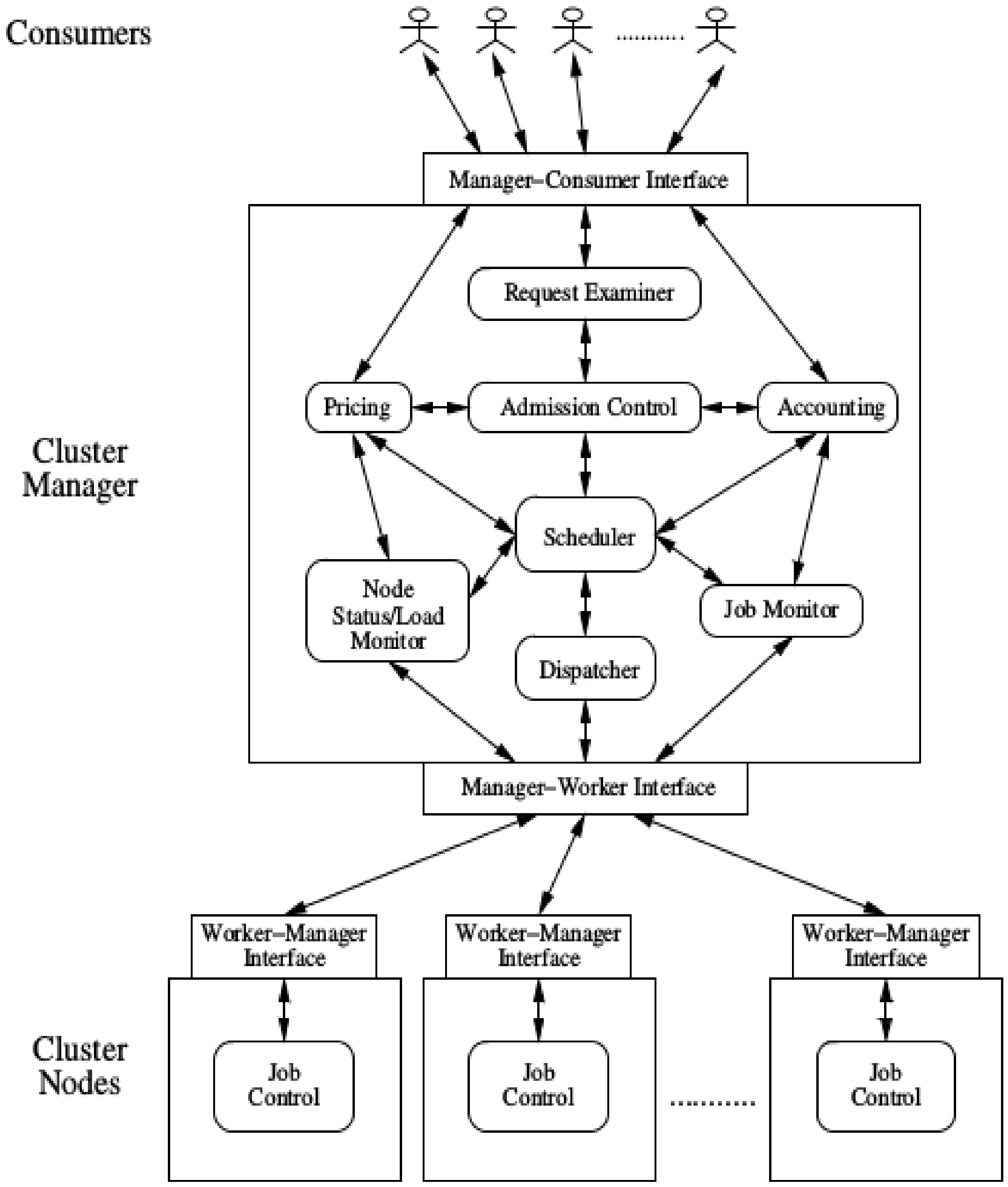

Figure 2.8: Abstract model for market-based cluster RMSs [35].

practice. The work in [35] also describes the models we will use and develop in chapter 3. It provides the economic models that establish how resources are allocated in a market-driven computing environment. In the commodity market model, producers specify prices and consumers pay for the amount of resources they consume. The pricing of resources can be determined using various parameters, such as usage time and usage quantity. In addition, the Posted Price model operates similarly to the commodity market with the difference that special offers are advertised openly so that consumers are aware of discounted prices and can thus utilize the offers. Finally, the auction model describes a way to allow multiple consumers to negotiate with a single producer by submitting bids through an auctioneer. The auctioneer acts as the coordinator and sets the rules of the auction. Negotiation continues until a single clearing price is reached and accepted or rejected by the producer.

2.5 Economic models for resource management and scheduling in Grid computing [8]

2.6 Motivation

As reliability becomes a more important factor in Many-core platforms, there is a need to manage the workload at run-time. By managing resources with reliability-aware methods, significant steps can be made towards prolonging the lifetime of Many-Core processors. Market-based resource management techniques have been developed in related work with the application field reaching to distributed systems, clusters and to the cloud. Our objectives regarding this diploma thesis are to propose models that will be evaluated with respect to reliability and to develop a resource management framework for Many-Core on-chip systems that will support these models under market principles. Besides that, the developed framework will provide a solid basis for further development in run-time resource management for Many-Core.

Chapter 3

Proposed Methodology

3.1 Introduction

Our proposed methodology is a run-time, distributed resource management framework. Our interest is in applying different market-based models to enable resource management. The framework is responsible for providing a way that the incoming tasks are initially mapped to available resources as well as providing the infrastructure of managing the task throughout its execution lifetime. The market models developed view application tasks as buyers and resources as sellers and try to create matchings amongst them. This framework is developed with intention to minimize system reliability degradation and as a result to distribute workload accross the cores in a way that core reliability degradation is uniform. Our approach takes into account both its workload and the condition of the system resources. So, in essence we create a market where the products for sell are the various types of resources offered by the platform and the buyers are the various tasks that want to run on this platform.

3.2 Resource Degradation Overview

Mean Time To Failure (MTTF) is a quantity that measures average time for failures to happen with the modeling assumption that the failed system is not repaired. Failures in complex, repairable systems are considered conditions not resulting from the design, that bring the system out of service or in need of repair. Any other failures that can be left or maintained in an unrepaired condition are not considered failures under this definition. Failures on Processor chips can be transistor circuit failures and delay faults caused by decreased transistor switching speeds due to NBTI-induced transistor aging [1].
In [13], it is demonstrated how system reliability is affected by the runtime phenomena on chip. More specifically, it is presented that an increase in the number of tasks with fixed execution time (i.e. an increase in system utilization) causes a decrease in MTTF as well as an increase in chip temperature. Consequently,

$$MTTF \propto \frac{1}{utilization}$$

The different models developed in the proposed framework will be evaluated with respect to the above relation, using MTTF as a metric.

3.3 Resource Modeling

The gradual degradation and healing characteristics of some reliability mechanisms allow resources to extend their operational availability and lifetime by carefully managing their workload and operating conditions [1]. However, reliability degradation at the chip level is expected to become worse for future platforms, as chip technology continues to scale down.

The proposed framework uses a distributed way of evaluating the resource conditions. Processor cores can evaluate their condition and give out an indication, so that the run-time manager can make decisions regarding resource allocation. Individual resources would like to keep reliability degradation as low as possible, but this goal also leads to conflicts with the processing power of the platform. Therefore, the run-time resource management approach should deal efficiently with this tradeoff and be compatible with both the expectations of application tasks(i.e. timely execution) and the expectations of resources(i.e. prolonged lifetime).

To be able to evaluate the core condition, some performance counters are used. By being able to monitor each core's running idle time and running active time(time of workload execution up to the time of the measurement) we can calculate the core utilization and consequently have a quantitative result of MTTF, since they share proportionality with utilization, as explained in section 3.2.

Next, we will use market terms to create a price model which will define the resources' availability and ease of participation in the resource allocation process, as well as its behavior over time.

Resources behave as sellers in a market as they provide services to buyers(i.e. application tasks) in the form of execution time. The resource ask prices reflect their availability or capability to execute application tasks, which can be hindered by operating conditions, such as increased temperature, or by failures that happen due to performance degradation due to NBTI for example. By disregarding these factors resources might exhibit permanent failures and unavailability as well as critical degradation of the platform processing power. For this reason, resource ask price has a threshold P_t, above which the resource is forced to remain idle until the ask price returns to acceptable levels, in order to protect it against the risks mentioned above.

The resource price behavior is defined by the following function:

$$P_{R_U}(duration, idle, match, P_{R_O}) = \begin{cases} P_{R_O} + duration & \text{, if } P_{R_O} < P_t \\ & and \text{ idle=1 } and \text{ match=1} \\ P_{R_O} - duration \times \Delta Price_R & \text{, if idle=1 } and \text{ match=0} \\ P_{R_O} & ,otherwise \end{cases}$$

The Price Update function has the following domains for its inputs:

$$P_{R_U} : (\mathbb{N}_{>0}, \{0,1\}, \{0,1\}, \mathbb{N}_{>0}) \rightarrow \mathbb{N}_{>0}$$

Everytime the run-time manager runs to perform resource allocation, the Resource prices are recalculated. Therefore, P_{R_O} denotes the Price of the resource after the previous price calculation an the result of the function is the updated price.

In the first case a resource is under the price threshold, its state is idle and it getting allocated to an application task, after becoming matched to that. After the task execution, the resource price is incremented by the application task duration. In the next case, the task is idle and is not allocated to any application task. For this reason, during the time the resource remains idle, its price decreases as the resource goes through a period of relaxation. In, this case, the Relaxation is expressed in the form of duration. The $\Delta Price_R$ is a fixed parameter that expresses the sensitivity of this price. Resources are given an initial starting price at power-up. Since resources are homogeneous (considering no workload has been executed) the initial price is identical in all resources. In the last case, the resource price remains unchanged, since the resource is either busy executing a job or above threshold and the resource is forced to become idle.

3.4 Task Modeling

Application tasks enter the resource manager during run-time, in order for them to be executed. In our approach application tasks behave as buyers in a market. Individual tasks would like to be executed as soon as possible and for this reason each task has an offer price that makes available in the market. When the task's offer price is equal or greater than the ask price of a resource, the task is matched to the resource and ensures execution time.

Next, we will create a price model which will define the task's priority and ease of matching with an available resource in the resource allocation process, as well as its behavior over time. Application tasks behave as buyers in a market as they offer to pay a price to sellers in exchange for their services in the form of execution time. The task offer prices reflect their urgency to be executed, which is typically linked with approaching deadlines. Tasks set their offer prices based on their individual status and expectations.

The resource price behavior is defined by the following function:

$$P_{TASK}(waiting_time, P_{TASK_O}) = P_{TASK_O} + waiting - time \times \Delta Price_T$$

The Price Update function has the following domains for its inputs:

$$P_{TASK} : (\mathbb{N}_{>0}, \mathbb{N}_{>0}) \to \mathbb{N}_{>0} \tag{3.1}$$

Everytime the run-time manager runs to perform resource allocation, the Task prices are recalculated. Therefore, P_{TASK_O} denotes the Price of the task after the previous price calculation an the result of the function is the updated price.

The waiting-time parameter is the number of periods each job is waiting until it is executed. This parameter is involved in order to express the urgency of execution of the application task. As the time comes closer to the deadline and the task has not yet been matched to a resource, its price increases so that the probability of a match in the next period is elevated. The $\Delta Price_R$ is a fixed parameter that expresses the sensitivity of this price.

Application tasks are given an initial starting price on entering the run-time resource manager. This price depends on the task's deadline and on an estimation about its duration.

This initial price is used to sort application tasks according to their relative priority. The price is given by the following relation:

$$P_{TASK_0} = P_T \cdot (50\% \cdot deadline + 50\% \cdot duration_{est}) \tag{3.2}$$

The initial application task price is the average of the normalized deadline and duration estimation. The outcome is a percentage which is later multiplied by the threshold resource price P_T. This is the maximum price a task can receive initially. At this price, any job would be instantly matched with highest priority to any available resource.
Regarding the benchmarks used in the framework, 10 benchmark classes were created. These benchmark classes differ in the duration and deadline. More specifically, each benchmark class has duration and benchmark that lie on a different percentile. For example, the duration of the first class has a duration in the lowest 10% of the maximum duration, the second class has a duration between 10% and 20% of the maximum duration, etc. The workload created is focused on the duration that an application task occupies the resource (i.e. the pipeline of a core), as the models proposed are aiming at the sceduling/mapping of application tasks on resources.

3.5 Proposed Market Models

3.5.1 Introduction

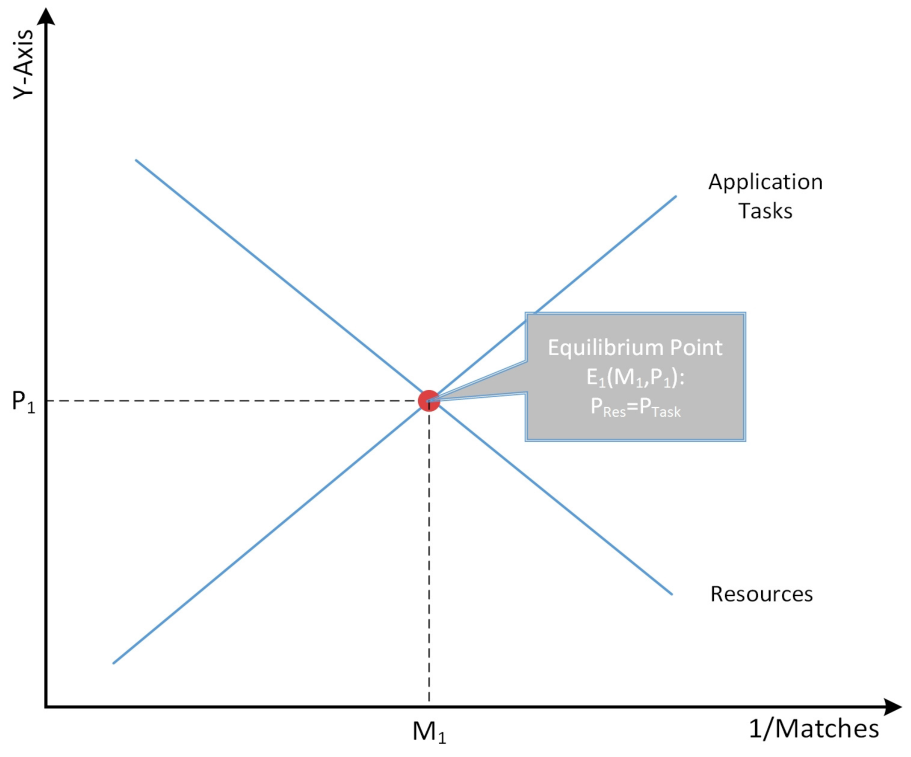

Figure 3.1: Supply/demand equivalent for tasks/resources

After modeling both the resources and the application tasks according to market principles we would like to observe, how fluctuations in the price of the tasks or resources affect the market. According to the functions defined previously, we can illustrate the resource and task prices in a supply-demand diagram (Figure 3.1). An equilibrium point reflects the specific price at which resources and application tasks match and is inversely proportional to the number of tasks matched to resources per time period. When the number of matches is low, resources have a low utilization at this period and their ask prices are low. However, when a low number of jobs are executing, the remaining jobs' average offer price increases. Similarly, when the number of matches is high, resources have a increased utilization at this period and their ask prices are increasing and a large number of jobs are executing. Therefore, the remaining jobs' average offer price decreases. This behavior is exemplary of the natural tendency of this technique to reach an equilibrium. This is a dynamic equilibrium that may experience shifts from period to period. This effect is displayed in Figure 3.2, where changes in resource reliability alter the equilibrium position. More specifically, on a system that has reliable resources, their prices

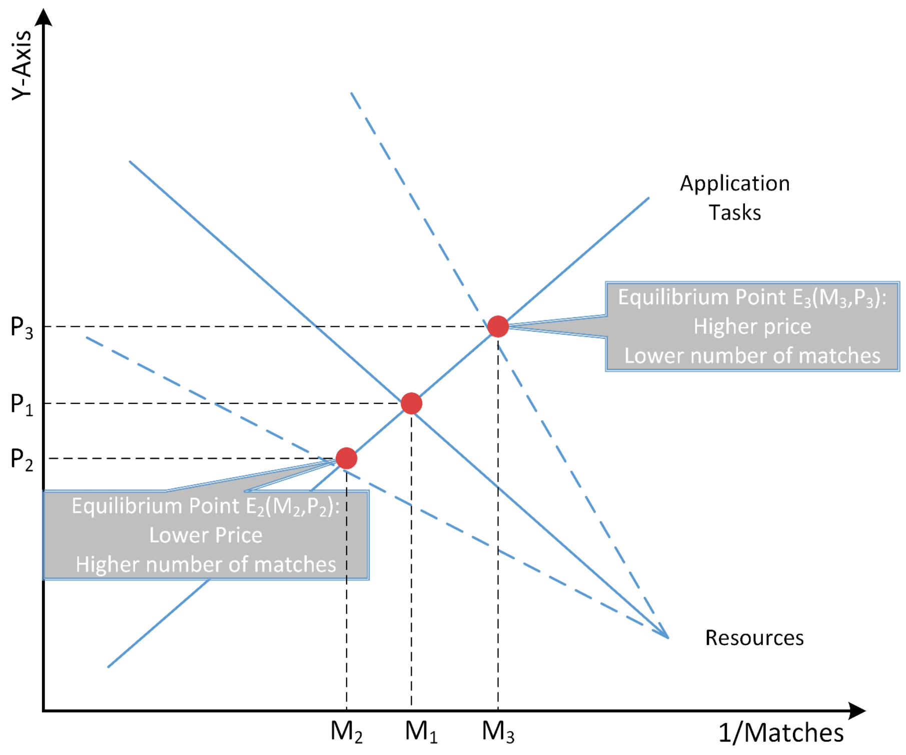

Figure 3.2: Impact of reliability fluctuations on equilibrium

decrease and a new equilibrium point E2 indicating higher number of matches per period is depicted on the figure. Reliable resources can execute more jobs. Similarly, on a system that has resources with low reliability, their prices increase and a there is a shift towards a new equilibrium point E3 indicating a lower number of matches per period. In cases that resources are overstressed and overutilized, resource prices may reach the price threshold

and become unavailable. In extreme cases, the number of matches per round can even reach 0.
Three different models are presented, namely the Commodity Market, the Posted Price and the Auction models. The concepts of these models have been initially presented in [8] and have been adapted to fit in this Many-Core system implementation. These models perform a matching in every time period the runtime framework makes resource management decisions. The matching is performed in a market created by available resources and incoming application tasks. The market between resources and tasks can be easily visualized as a bipartite graph whose nodes are resources and tasks and whose edges are matchings(Figure 3.3).

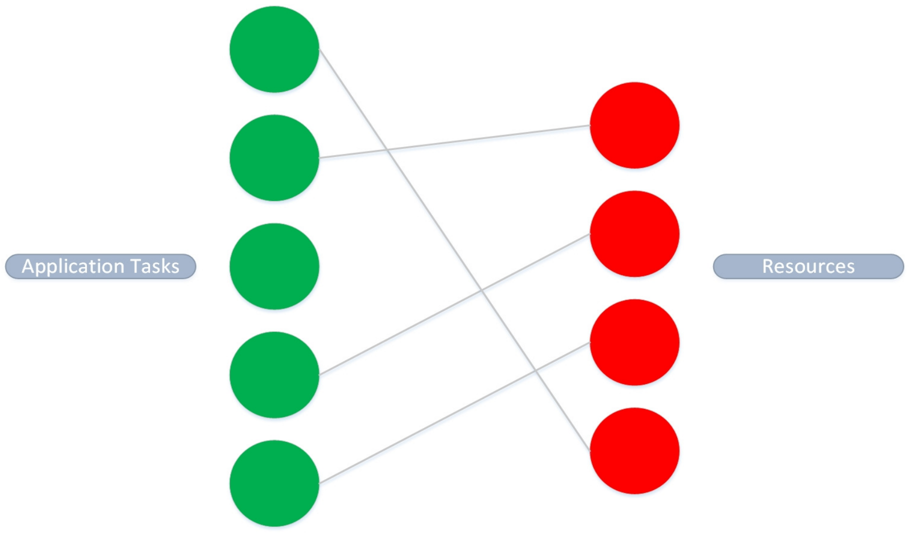

Figure 3.3: Market of available resources and application tasks

3.5.2 Commodity Market Model

According to [8] the definition of the model is the following:
In the commodity market model, resource owners specify their service price and charge users according to the amount of resource they consume.
In this approach, when the price offered by an application task satisfies a specific resource, a match is found and the task is assigned for execution to the given resource. In general, execution time is priced in such a way that supply-and-demand equilibrium is maintained. The behavior of this model is illustrated previously in Figures 3.1 and 3.2.
Both available resourses and application tasks are located in queues as illustrated in Figure 3.3. Matching between application tasks and resources is performed with the following steps:

1. All application tasks and available resources are sorted in descending order according to their offer and ask price respectively and positioned into queues.

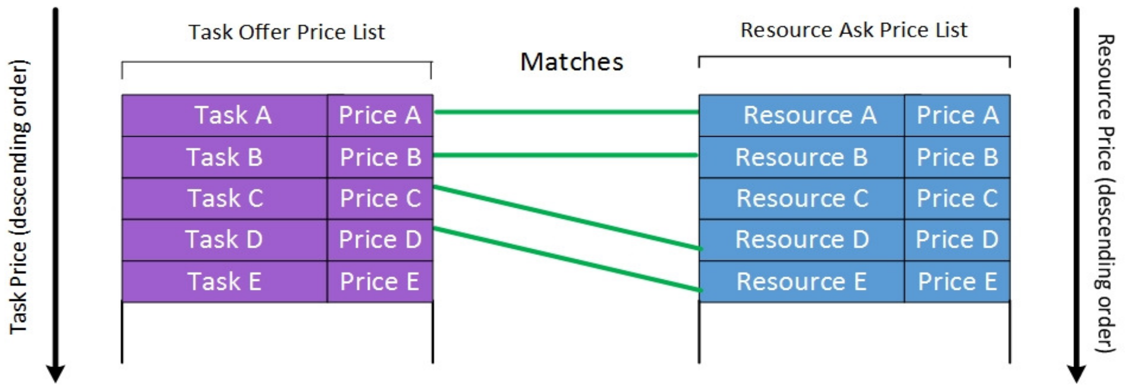

Figure 3.4: Resource and task matching

2. The highest priority task traverses the list of resources until its offer price is higher than a resource's ask price. When this occurs, a matching is created and the next iteration starts checking for matches from the next task and the next resource in the list. If a match is not possible, then the algorithm checks if the next resource in the list is a good match.

3. The algorithm iterates through the whole application task list to create all possible matches.

4. The algorithm stops if it reaches the end of either the resource list or the application task list.

In Figure 3.4 there is an example of the execution of the algorithm. The green edge denotes a matching, whereas the red edge denotes that no matching was possible as the resource ask price is higher than the task offer price. The matching algorithm is the same with the previous model, but the model offers an extra feature. When the framework needs increased utilization in order to cope with the increased number of jobs, this model initiates a discount period, where resources' price is lowered in order to increase their availability and ease of getting matched to tasks.

3.5.3 Posted Price Model

According to [8] the definition of the Posted Price model is the following:
The posted price model is similar to the commodity market model, except that it advertises special offers in order to attract consumers to establish market share or motivate users to consider using cheaper slots.
In this model, the basic principle of the Commodity Market model holds. That is that a match is found and an application task is given for execution to a resource when the offer price of the task is greater than the ask price of a resource. A graphical explanation of the discount mechanism is displayed in Figure 3.6. In this model the market enters a discount

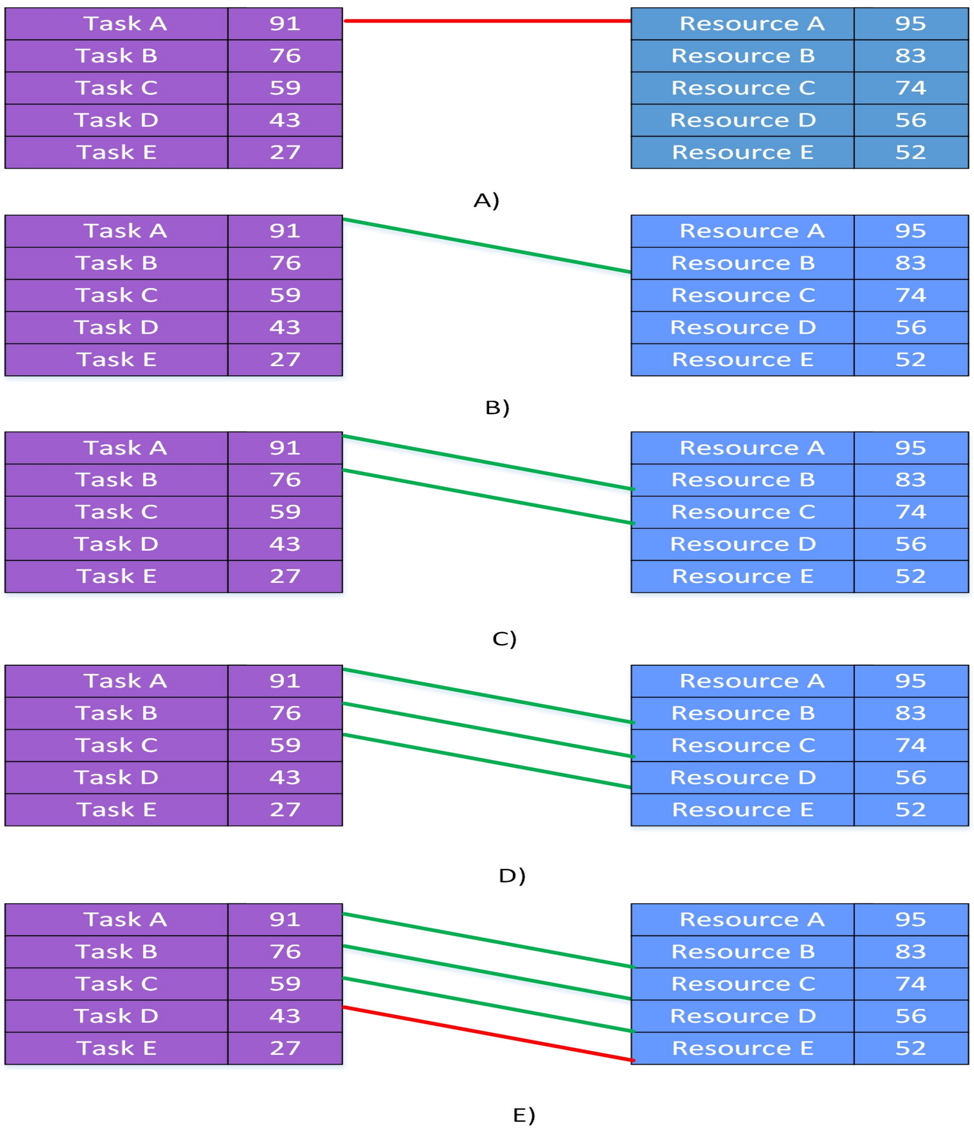

Figure 3.5: Matching example in the Commodity Market model

period if the unmatched application tasks become greater than the number of resources. During the discount period, resources have a discounted price. This discount period terminates, as soon as a predefined number of application tasks has been matched to resources that have reduced prices. After the end of this discount period, it is prohibited to initiate another discount period, before waiting for that predefined number of application tasks to be matched under regular prices. Discount periods and periods of discount prohibition need not necessarily be the same in terms of duration, as the market conditions affect the matching capacity. Consecutive discount periods are prohibited, because the market equilibrium is not dynamically adjusted during these periods, but is rather biased due to the resource ask price reduction. Continuously operating under the stress of the discount period, the system utilization and consecutively the MTTF will be affected.

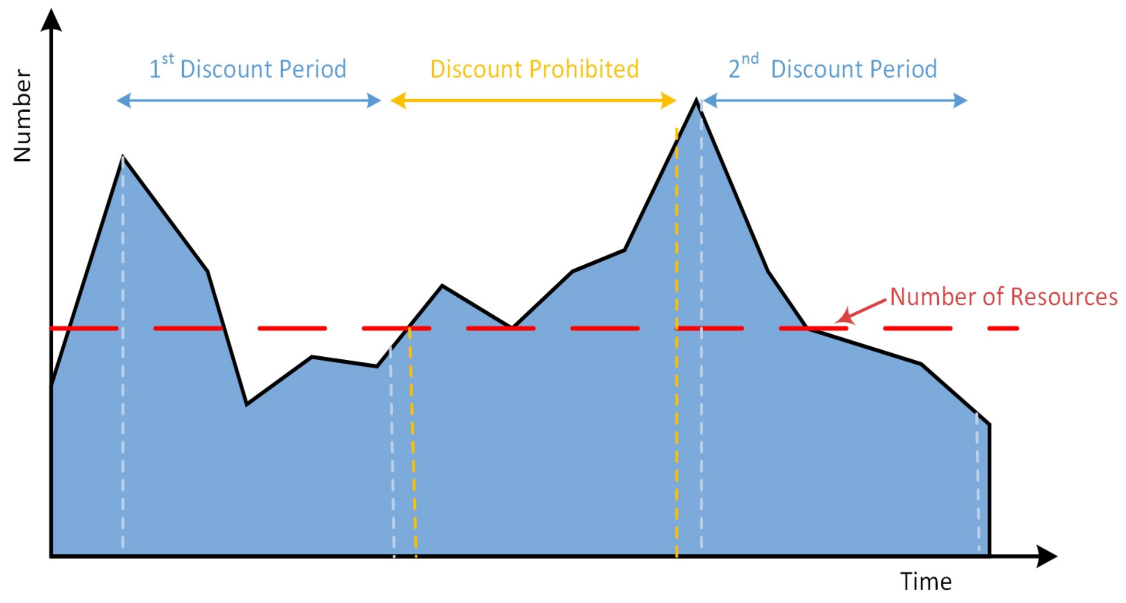

Figure 3.6: Discount mechanism of the Posted Price model

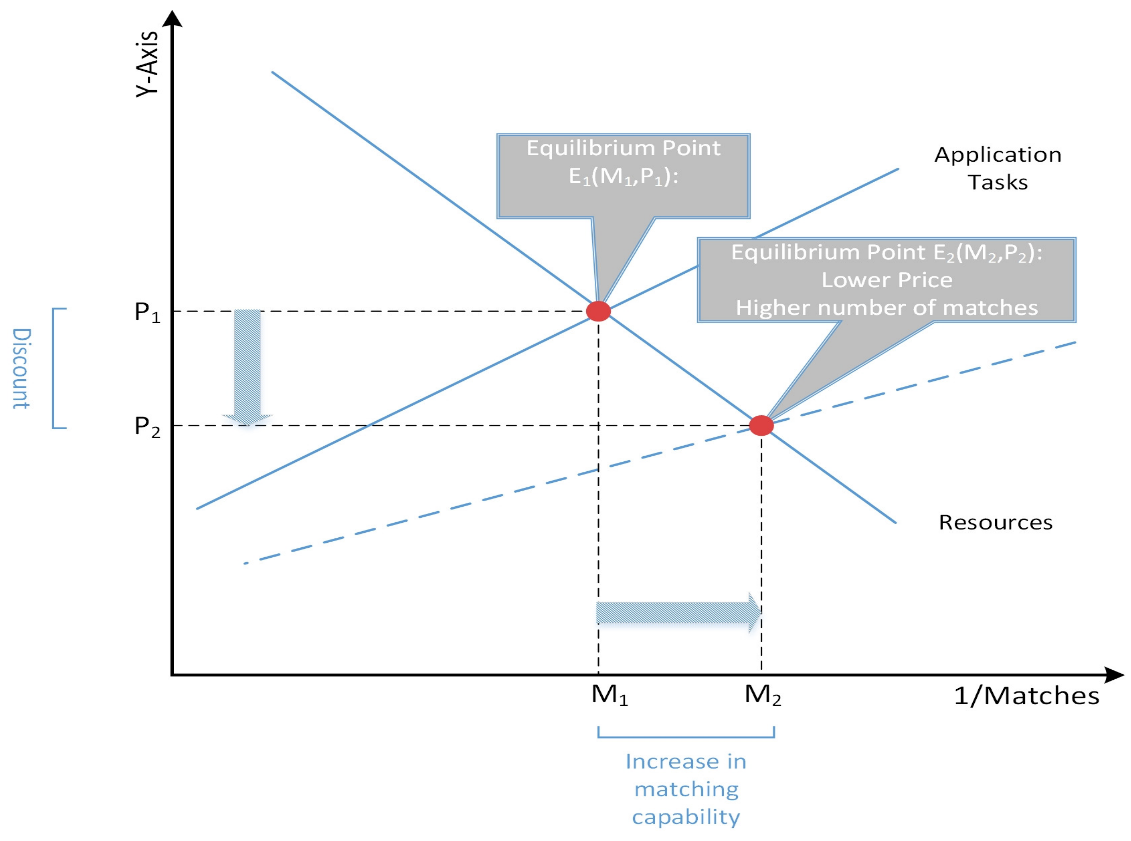
Y-Axis
Equilibrium Point E1(M1,P1):
Application Tasks
Equilibrium Point E2(M2,P2): Lower Price Higher number of matches
Discount
P1
P2
Resources
M1
M2
1/Matches
Increase in matching capability

Figure 3.7: Behavior of the Posted Price model

3.5.4 Auction Model

Overview

According to [8] the definition of the Auction model is the following:
The auction model supports one-to-many negotiation, between a service provider (seller)

and many consumers (buyers), and reduces negotiation to a single value (i.e. price). The auctioneer sets the rules of auction, acceptable for the consumers and the providers. Auctions basically use market forces to negotiate a clearing price for the service.
For this approach, the Sealed-Bid First-Price Auction scheme has been used. In this auction type, bidders conceal their bids and submit them. The person submitting the highest bid wins the object and pays what he bid. Specifically, the auction mechanism designed functions in a distributed function. Every resource creates a bid according to their own expectation, evaluation and to the objective value of the task(based on execution duration and deadline) and submits it.
To model the auction procedure, we have a market that consist of resources (i.e. potential buyers) and application tasks(i.e. goods or services that are about to be auctioned). For every different application task that is auctioned to the resources, a different market is created. Application tasks are in a queue and the task that is located on the head of the queue enters the market and is presented to the available resources and a matching occurs after an auction. Every resource has two parameters that affect its bidding price.

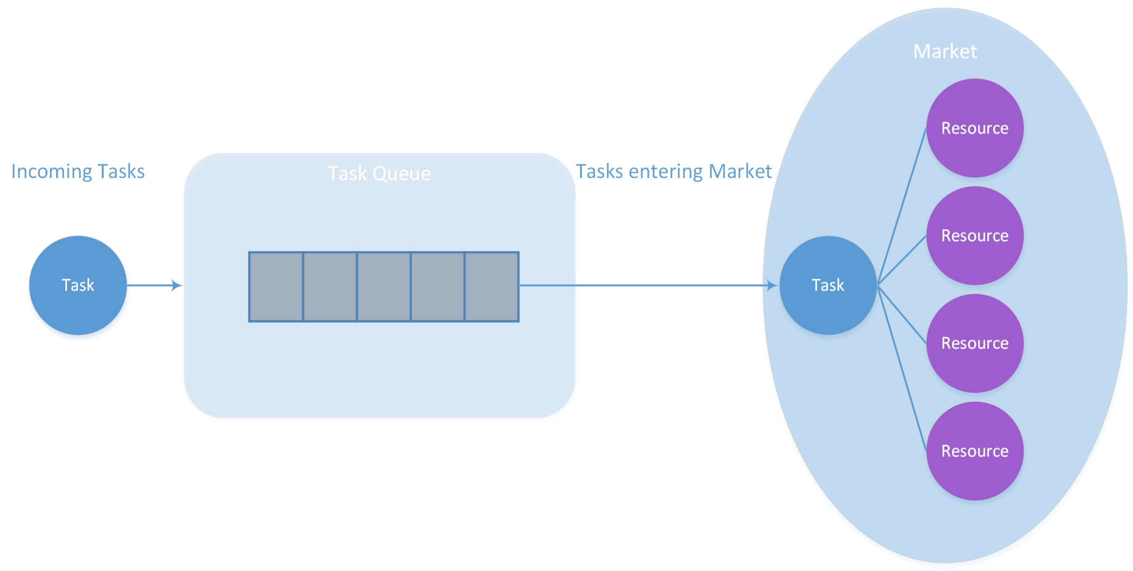

Figure 3.8: Task queue in auction model

1. The total utilization of the resource as a fraction of the total execution time of application tasks on the resource to the total time elapsed, that includes the task execution time (T_{exec}) and resource idle time (T_{idle}).

$$utilization = \frac{T_{exec}}{T_{exec} + T_{idle}} \tag{3.3}$$

2. The duration of resource execution time requested from the application task.

In the following section, the expected behavior of the auction model will be discussed. Utilization and task execution duration behavior will be characterized and given equations

that apply to the supply and demand curves.
For our analysis, we assume a constant application task execution duration requested from the resource.

Resource utilization modeling

As the utilization of a resource increases, we expect this resource's price to increase as well. Additionally, the resource has more probability of becoming matched to an application task. Similarly, when the resource utilization decreases, the resource's price is expected to become less and the resource to become less likely to get matched to an application task. Therefore, resource utilization is proportional to the resource price.
There is an analogy to the economic model.
As the quantity decreases the price decreases as well, whereas in the case of an increase in quantity the price rises.
We can conclude that the utilization-price relationship has a behavior similar to the supply curve.

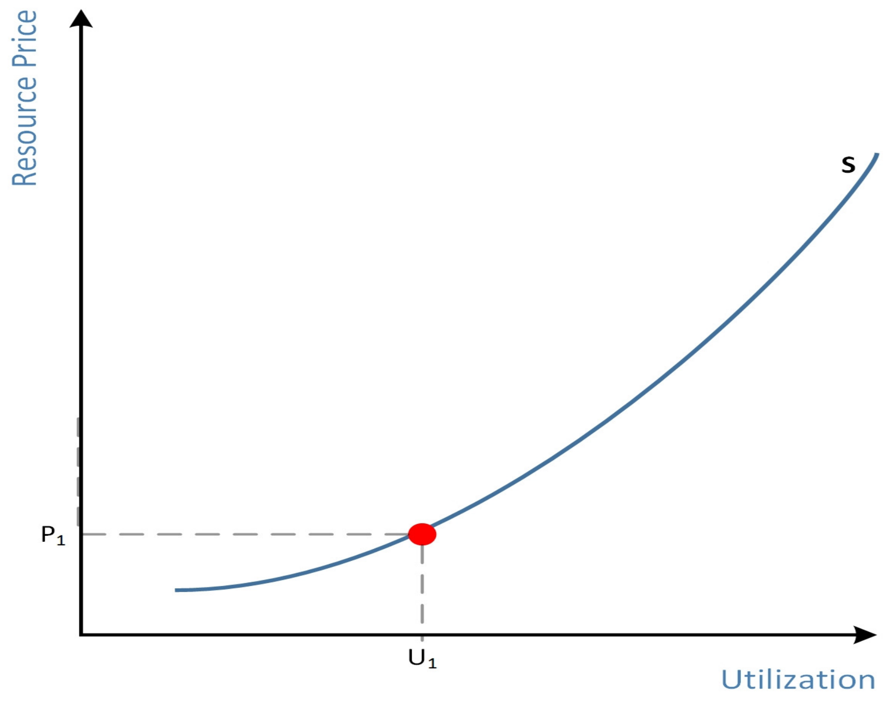

Figure 3.9: Supply-based utilization curve

Task duration modeling

In order to create a mapping of application tasks to resources that performs efficiently in terms of uniform resource degradation and balanced resource utilization, the assignment of tasks to resources needs to be as efficient as possible. In order to achieve that, every

core needs to have about the same T_{exec}. An indication could be the the number of tasks executed on each resource. However, the application tasks vary in duration and the matching would not be efficient. For this reason, the performance counters discussed earlier can provide a fine-grained evaluation of the total execution time of each core. The execution duration of an application task about to run on a resource is defined as $T_{duration}$.
The expected behavior is the following:
When a task $T_{duration}$ increases, this application task becomes more difficult to match to a resource and the resource price evaluation is high, whereas in the case of a low $T_{duration}$, the application task becomes easily matched and the resource price evaluation of the task is low.
Consequently, $T_{duration}$ is proportional to the Resource price.
There is an analogy to the economic model.
As the quantity decreases the price decreases as well, whereas in the case of an increase in quantity the price rises.
We can conclude that the task execution duration-price relationship has a behavior similar to the supply curve.

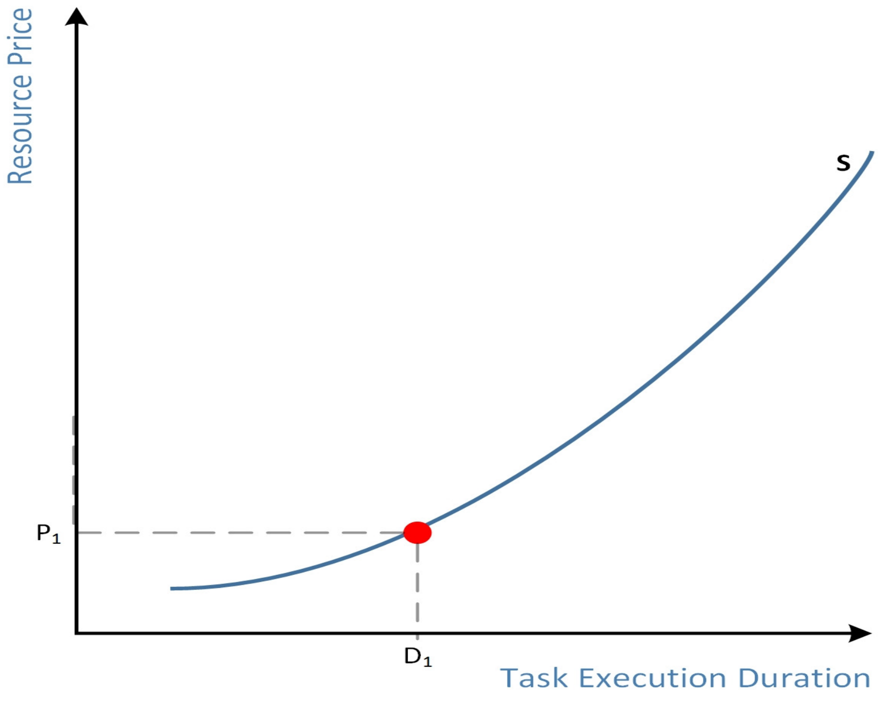

Figure 3.10: Supply-based task duration curve

Price extraction

A simplified equation for the demand curve is the following:

$$Price = \alpha \cdot Quantity + \beta \tag{3.4}$$

Since these resource utilization and task execution duration follow the demand curve:

$$ResourcePrice = \alpha \cdot utilization + \beta \tag{3.5}$$

$$ResourcePrice = \alpha \cdot T_{duration} + \beta \tag{3.6}$$

As we can see from the equations, these two parameters can provide a resource price. The smaller the price, the greater the probability for a task to be matched to a resource (We need the small prices to have that effect, in order for zero utilization to have zero price as an initial condition). Both resource utilization and task execution duration are proportional to the resource price:

$$ResourcePrice \propto utilization$$

$$ResourcePrice \propto T_{duration}$$

Since resource utilization and task execution duration have equations $(2), (3)$, the following product corresponds to an resource price indication for the current (current means that the application task has been executed and the T_{exec}, T_{idle} are up to date) period n and is denoted as f_n:

$$f_n = utilization \cdot T_{duration} \tag{3.7}$$

Based on (3.1), (3.4) becomes:

$$f_n = \frac{T_{exec}}{T_{exec} + T_{idle}} \cdot (T_{duration}) \tag{3.8}$$

In order for each resource to make an evaluation and consequently a bid, the knowledge of the task execution duration is necessary. However, the task duration execution is not known before the execution itself. Therefore, a prediction of the task execution duration T_{pred} will be used instead.
In order to calculate the resource price indication for the next period $n+1$ we will use the following:

$$T_{exec_{n+1}} = T_{exec_n} + T_{pred} \tag{3.9}$$

The execution time in the next period will include the execution time of the current period added by the predicted duration of the application task, which has not yet been included in the calculation. Therefore, the resource price indication will become:

$$f_{n+1} = utilization \cdot T_{pred} = \frac{T_{exec_{n+1}}}{T_{exec_{n+1}} + T_{idle}} \cdot T_{pred} \tag{3.10}$$

After the substitutions the resource price will be:

$$f_{n+1} = \frac{T_{exec_n} + T_{pred}}{T_{exec_n} + T_{pred} + T_{idle}} \cdot T_{pred} \tag{3.11}$$

If we knew all the real $T_{duration}$ instead of every T_{pred} for every application task, then we could assign the incoming tasks to resources with the lowest price without error. In every period, all the incoming application tasks are sorted according to their prices and enter the market in that order. By making an optimal assignment for every job auctioned, we manage to always assign jobs to the least utilized resource, achieving fairness and balancing between them.
By substituting $T_{duration}$ with T_{pred} we make an appoximation of the optimal matching by introducing some error due to the prediction.
In our framework, application tasks are entering the market one by one after their creation and are auctioned to the resources. In our framework, there are periods, in which more than one application tasks are created at a time. For this reason, application tasks are inserted in queues ordered by their price. Because our framework is a resource manager operating at runtime, tasks are placed in the market in their order, as reordering would be false, despite the fact that this might result in an overall better solution.

Task duration prediction

We want to create a mechanism that could predict the duration of the incoming application task based on the task's deadline, as the duration is not known a priori. For this reason, we will explain the implemented approach, based on the theory in [28].
We will present two methods with slightly different features. These are the Stochastic gradient descent method and the Gradient descent for linear regression.
The main idea behind these methods is that we would like to find a linear function $h_\theta(x) = \theta_0 + \theta_1(x)$ that has the least possible distance from the training samples (x_i, y_i). A function like that with a given training set is illustrated in Figure 3.11. Therefore, we define a cost function as following:

$$J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{3.12}$$

This is a function that returns the total distance from our training set for a given choice of parameters $\theta_0, \theta 1$. The goal is to minimize that function in order to find the optimal choice of parameters.
If we plot this function, we can see a three dimensional function that displays the cost in relation to the two parameters. In Figure 3.12, we have plotted the cost function based on the training set displayed in Figure 3.11. We can observe that the function has one global minimum to which we want our algorithm to converge. As this is a convex function, it only has one global minimum, no local minima and the solutions always return the same result.
After defining the cost function, we need to use a method to minimize the cost function. The general concept is that we start with some initial parameters $\theta_0, theta_1$ and keep changing these in order to reduce $J(\theta_0, \theta_1)$ until we manage to reach a minimum. We will present the two methods.

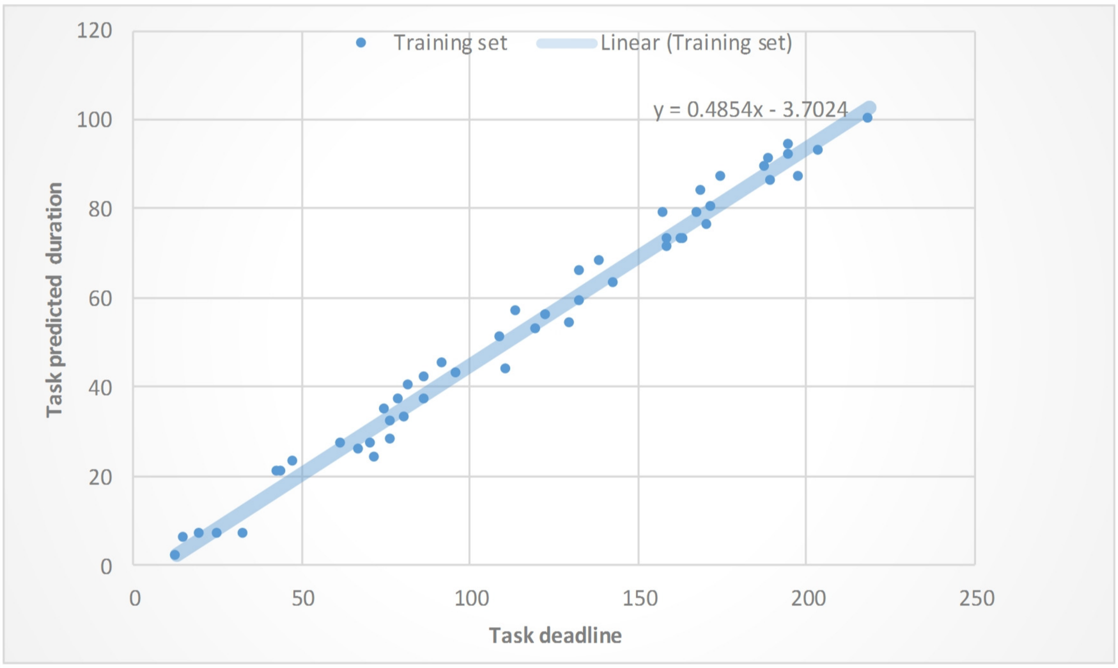

Figure 3.11: Optimal linear function produced by gradient descent

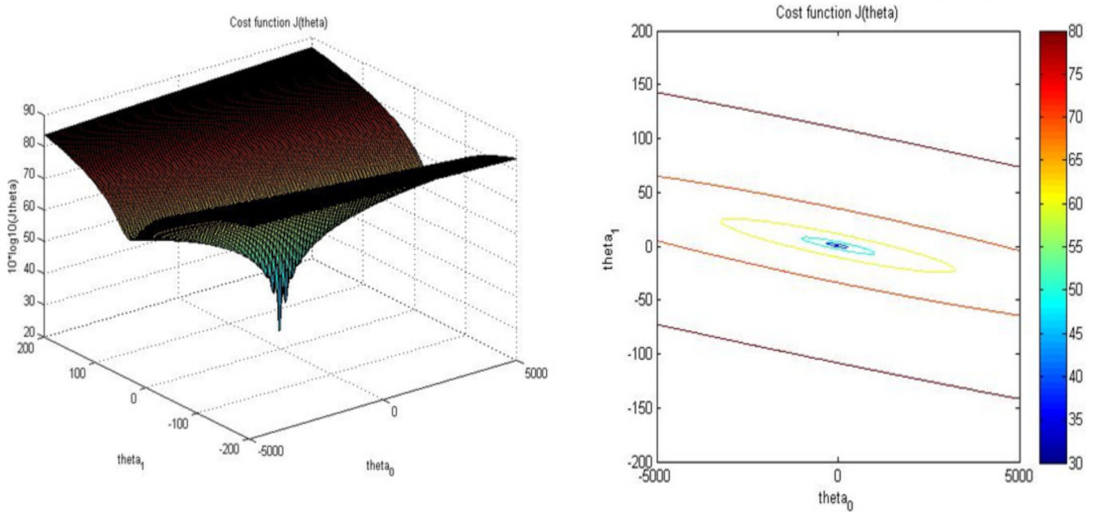

Figure 3.12: Cost function used by gradient descent

Gradient descent using linear regression

For this algorithm we need to repeat the following equation (for j=0 and j=1) until we reach convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \tag{3.13}$$

In the above equation, α is the learning rate. If we solve tha partial derivative the algorithm becomes the following:

Choose an initial vector of parameters θ_0, θ_1 and learning rate α.
Repeat until convergence {

$$\theta_0 := \theta_0 - 2 \cdot \alpha \cdot \tfrac{1}{m} \textstyle\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$
$$\theta_1 := \theta_1 - 2 \cdot \alpha \cdot \tfrac{1}{m} \textstyle\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^i$$

}

Stochastic gradient descent

The cost function in equation 3.12 describes a function in the following form:

$$J(w) = \sum_{i=1}^{m} J_i(w) \tag{3.14}$$

where the parameter w is to be estimated and where each J_i is associated with the i^{th} observation in the data set.
In stochastic (or "on-line") gradient descent, the gradient J(w) is approximated by a gradient at a single example [7]:

$$w := w - \alpha \nabla J_i(w) \tag{3.15}$$

The algorithm in this case becomes the following:

Choose an initial vector of parameters θ_0, θ_1 and learning rate α.
Repeat until an approximate minimum is obtained {

$$\theta_0 := \theta_0 - 2 \cdot \alpha \cdot (\theta_0 + \theta_1 x^{(i)} - y^{(i)})$$
$$\theta_1 := \theta_1 - 2 \cdot \alpha \cdot (\theta_0 + \theta_1 x^{(i)} - y^{(i)}) \cdot x^i$$

}

Prediction design choices

Stochastic gradient descent provides an approximate solution of the global(or local) minimum. However, in contrast to the standard gradient descent, it offers the advantage of being on-line. Especially, in this case, the framework developed aims to manage application tasks at runtime. Since the algorithm is on-line and much of the computation is performed in a distributed fashion, application task samples can be made available to the system while the computation of the coefficients is taking place. This means that the training set can grow at runtime. At simulation level, both algorithms offer the similar results, as there is not enough availble parallelism to populate the training set at runtime.
In order to keep the overhead of the prediction low, only a small number of iterations has been performed towards convergence. A certain range has been selected for parameters $\theta_0, \theta 1, \alpha$ by using the trial and error method. All the selected values ensure convergence for the training sets that the algorithm executes with.

Chapter 4

Implementation of proposed framework

4.1 Introduction

We developed a simulator for our market-based resource management approach in a common x86 system. The simulator provides the following:

i. A way to be able to represent the market models effectively and an infrastructure to base different models and algorithms on.

ii. A modular code base that enabled the development of different market models without introducing a great additional overhead.

iii. A way to get experimental results in order to be able to verify the functionality of the proposed framework.

iv. Experimental results validating the scalability of the framework in platform with a large amount of cores (up to 256).

A combination of the C/C++ programming languages was chosen for the development of the simulator. These offer both the low-level capabilities of C (system calls, handling of threads/processes, synchronization) and the advantages of object-oriented programming. The simulator ran on top of a standard Linux operating system. For this resource management framework three instances have been created, one for every different model. The framework architecture is illustrated in Figure 4.1 and includes the following:

- Job Creator is the basic component of the framework responsible for runtime.
- Market is the component where the economic models responsible for the resource management are implemented.
- Dispatcher is responsible for realizing the mapping decisions of application tasks to resources, performed in the market module.
- Resource Update is the module for making changes to the resource states.

4.2 Framework Architecture

4.2.1 Job Creator

The job creator is a basic component of this framework. It is responsible for the workload generation and for providing the market module with tasks and available resources. In addition, the job creator is the module responsible for the runtime, as it creates instances of the market in every run-time period.

The job creator manages files in order to keep the run-time system consistent and up to date. Some of these files are shared with other modules of the framework, the most important of which is the task queue. The job creator generates workload with specific deadline and duration, which vary according to the benchmark class that the application

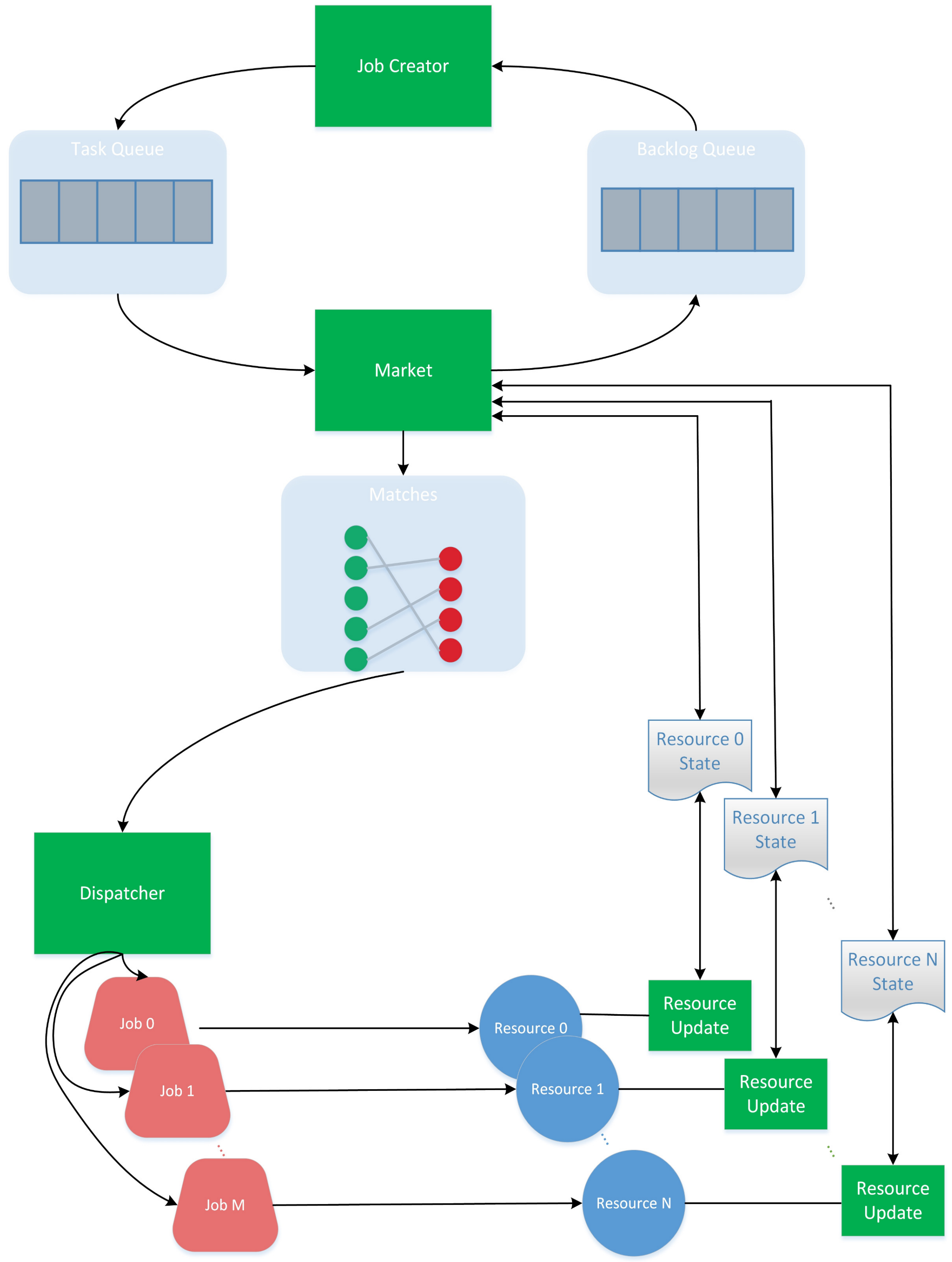

Figure 4.1: The Resource Manager Architecture

task belongs to.

The job creator also provides the capability of regulating some parameters of job creation. First of all, the frequency of job generation can be defined by assigning a specific proba-

bility to it. During these task creating periods the number of newly created jobs varies. The range depends on the number of resources that the particular instance of the run-time system has. Additionally, there are rare occations, when the job creator produces a job burst, when a large number of jobs floods the run-time system.
There are two queues of importance on the job creator. First, there is the task queue that holds all the newly created tasks together with old tasks from previous execution periods that haven't yet been matched. All these are put together and sorted according to their offer price in descending order (priority). The second queue is the backlog queue that holds all unmatched and undispatched tasks from previous execution periods. Before these tasks are put together with the newly created tasks, their offer prices are updated according to the equation explained before.
This module keeps running, until no further job is created and all tasks have finished their execution.

4.2.2 Market

The Market module is the core of the market-based resource management framework. The main functionality that it implements is the matching of application tasks to available resources and the initiation of the dispatcher module. This module loads all task and resource information and places it in two different lists. These data structures enable easy access to all necessary information and facilitate the matching. For every different instance of the framework, a different model is implemented. After the model-dependent matchin, the matched pairs are put into a log, which is later required for the dispatcher. After the matching, the backlog queue is filled with the application tasks that could not find a match.
Normally, the task prices are defined in the job creator module. However, for the auction model another functionality is implemented. A distributed parallel price calculation mechanism is implemented, which enables individual resources to make their own evaluation of the task price, given its duration prediction. The resources then return the prices to the master(controller) resource, who then selects the resource with the lowest price.
Moreover, the auction model supports a prediction mechanism that executes the stochastic gradient descent algorithm to calculate the coefficients of the linear regression that will be used to predict the duration based on the deadline.

4.2.3 Dispatcher

The dispatcher is the module responsible for making the application tasks execute on the resources. It is initiated after the market module. The dispatcher receives a list of the matches and puts the application tasks for execution. It handles the application tasks throughout their lifetime. For this reason, a monitor-process is created for each application task about to execute. This process serves as an interface between the task and the modules, as it places it on the resource to run and updates both the task and resource states when necessary. The dispatcher also provides the infrastructure for the simulated workload to execute and waits for it to end to timely perform the necessary updates to the resource state files, making them available again. The mechanism that supports that will be explained in the sequence. The dispatcher also is responsible for all synchronization issues regarding processes as well as for keeping track of the execution period and being a reference for all these processes. The synchronization implemented for managing the run-time of processes is signal-based. Finally, after completion of the dispatcher execution, the job creator is notified to proceed to the next execution period.

4.2.4 Resource Update

The resource update module is responsible for making changes to the resource state files when necessary. These state changes occur when an application task is dispatched to a resource and when the task finishes its execution and the resource needs to become available again.

4.3 Design Decisions

In this section the design decisions we made in order to construct and code our proposed resource management framework are presented.

4.3.1 Clock Cycle Abstraction

In this framework, the run-time resource management is performed in execution periods. The proposed approach is a time period, which is independent of architecture and of actual time. Therefore, the abstraction of a clock cycle has been used, one that does not have timing specifications, but is only used for our simulation purposes. This clock cycle abstraction has been designed to evaluate the different models comparatively and to lessen the dependencies of the system, on which the simulation runs. Because this system cannot offer enough parallelism and resources like a Many-core system, the overhead of running parallel tasks in a serial way would make the simulation results unclear and the actual resource management intelligence developed difficult to distinguish from the results.

4.3.2 Resource States

Essentially there are only two distinct categories amongst the resources of a platform. A resource can either be the controller core or a common resource. In this implementation the resource mapping is performed in a centralized way and there is only one master resource, on which the job creator, market and dispatcher modules run. Every other common resource is responsible for executing workload and for performing resource state changes via the resource update module when necessary.
Resources can have three different states, which are presented below:

1. Busy state
   In this state, an application task gets execution time from a resource. An application task is matched with the resource inside a market and the resource is then exclusively allocated for the application task to run. For the whole duration an application task occupies the resource, the resource is set to the Busy state and all other application tasks cannot have this resource allocated. The resource does not partake in the market. While remaining at this state, resources have a fixed price, but after the application task finished execution, the task duration is added to their price.

2. Idle state
   In this state resources are available to application tasks. Idle resources are identified in the market module and are participating in the matching procedure, regardless of the outcome of the match. While remaining at this state, resources continuously decrease their price, because of the relaxation (decrease in utilization).

3. Forced idle state
   In this state, the price of the resource has increased above threshold. Therefore, even though the resource might be idle, it cannot execute an application task, as it needs to reach a price bellow threshold to do so. While in this state, resources continuously decrease their price, because of the relaxation (decrease in utilization). Below is a finite state machine displaying the transitions from one state to the other.

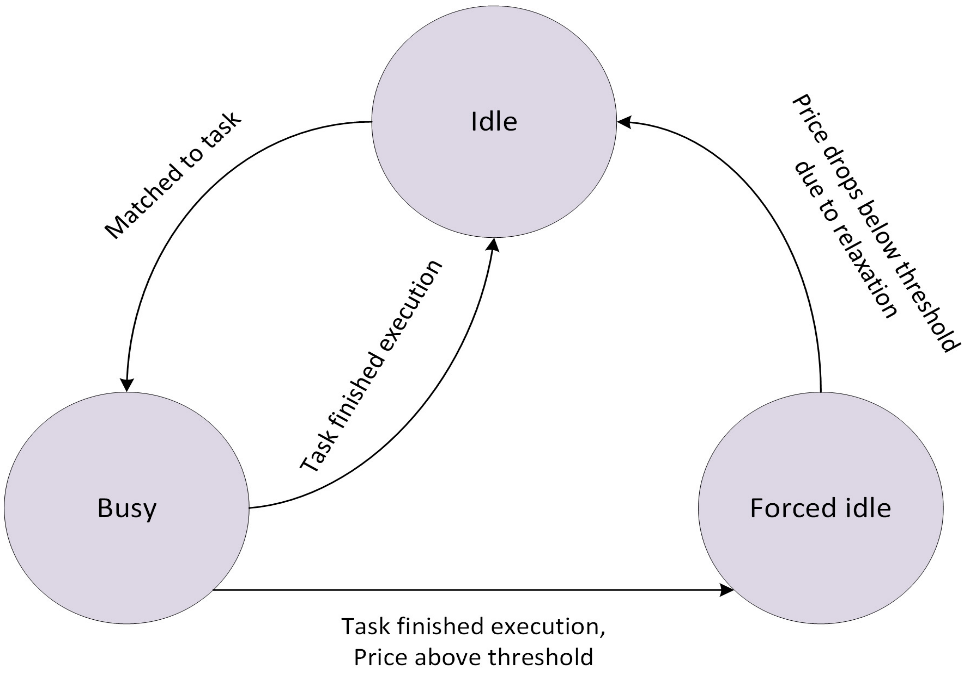
Idle
Matched to task
Price drops below threshold due to relaxation
Task finished execution
Busy
Forced idle
Task finished execution, Price above threshold

Figure 4.2: Resource state transitions

4.4 Implementation Details

4.4.1 Synchronization scheme

In the dispatcher, we implemented a synchronization scheme in order to enable the framework to manage application tasks throughout their lifetime. The dispatcher creates a process for every different application task and keeps it alive for the whole duration of its lifetime. For a duration of multiple clock cycles, the dispatcher employs a signal-based start/stop scheme. More specifically, after the workload executes its first cycle, a SIGSTOP signal is sent by the master resource to make it halt. In the sequence, after a new clock cycle begins a SIGCONT signal is sent to all resources executing a task to resume their execution. Then, all these tasks either execute workload if they are supposed to execute for a longer duration or terminate and return.

4.4.2 Distributed price calculation

For the auction model a price calculation is required for all resources and for every task that enters the market. This price calculation is required to be performed in a distributed fashion in order to reduce the computational burder on the master node, but also to achieve parallelism. The mechanism designed works as follows:

1. The master node gets the number of available resources.
2. The master node sends to all common resource the value of T_{pred}. T_{pred} is the predicted duration of the application task currently in the market.
3. Every resource calculates its ask price based on T_{pred} and its individual utilization.
4. When this value becomes available, it is returned to the master node.
5. When the master node becomes informed of the number of available ask prices, it receives these values and moves to the matching.

For every resource, a thread needs to be created for each available resource in order to send T_{pred} to a resource and request the ask price from it. In order for this to happen correctly, a thread synchronization mechanism based on mutexes is required. This mechanism has the following functionalities:

- While idle, resources receive task duration predictions and calculate their own ask prices based on that. During this price calculation, the resource's thread mutex remains locked.
- When the price calculation is complete, thread mutexes of the cores become unlocked, while making their ask price to the master node. When they become unlocked, they increment a counter that declares the total number of available cores.

- If resources don't become matched or locked, these calculate a new ask price for every different job that enters the market.
- Whenever a resource gets matched, its mutex that is currently inside the market becomes locked.

Finally, there is a condition for the master node to miss a price calculation. This can happen if the slave resource has not yet declared its availability, although the price has been calculated. Therefore, its mutex remains still locked. If this is the case, then this resource will become available and declare its availability in the next clock cycle.

4.5 Conclusion

In this chapter, we presented the architecture of the resource management framework and provided an overview of some important design choices. Finally, some implementation details have been explained, giving an insight to the synchronization issues and to the mechanisms that help simulating a distributed resource manager.

Chapter 5

Experimental Results

In this chapter, we will evaluate our simulation results. The simulation was performed for a variety of platform sizes varying from 8 cores to 256 cores. The evaluation includes the behavior of the different economic models over time, their performance in terms of execution time and their efficiency in load balancing, which is indicative of the reliability degradation (MTTF metric).
In the following two figures, the behavior of the Commodity Market model is illustrated. We can observe that the resource utilization of the system is proportional to both the backlog queue size as well as the number of incoming tasks. The resource utilization changes according to the fluctuation of the number of incoming jobs and jobs remaining in the backlog queue (Figure 5.1). In the next figure (Figure 5.2) we can observe the fluctuation

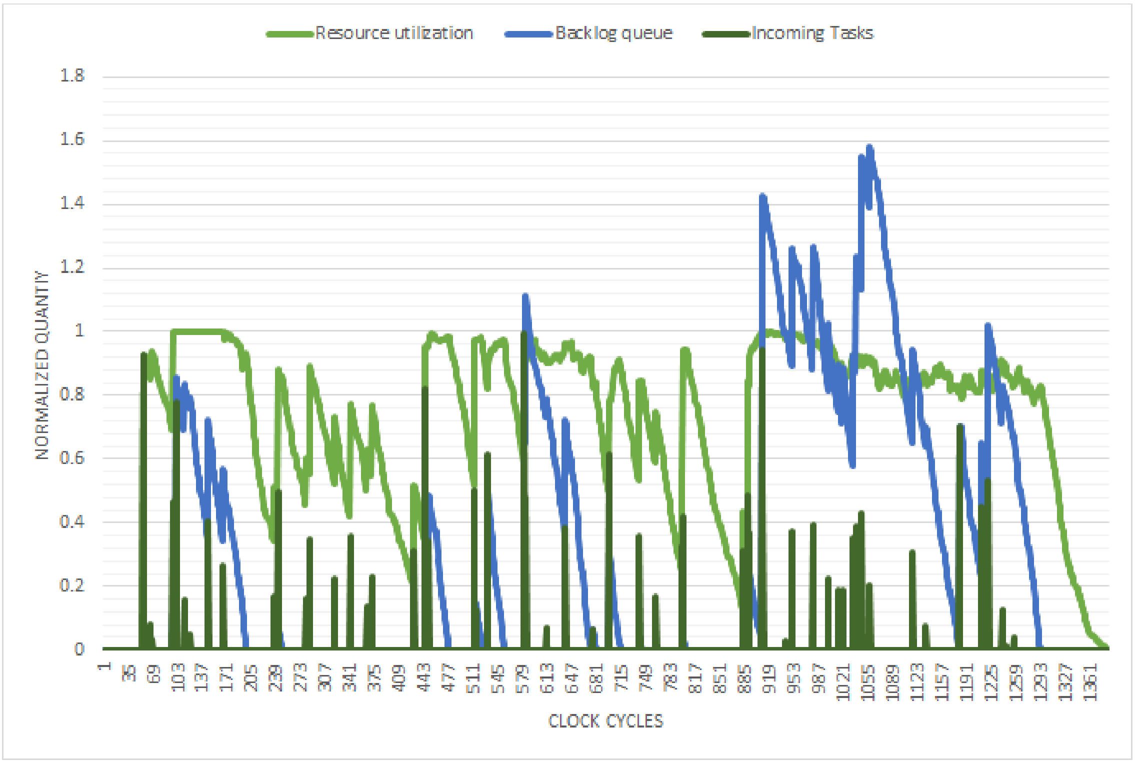

Figure 5.1: Behavior of Commodity Model over time

of the resource and task average prices. We can observe that the task average price is proportional to the backlog size. In addition, the resource average price is proportional to the resource utilization. The behavior of the Posted Price model is similar to that of the Commodity Market model, with the difference of discount periods. When discount periods are enforced, the utilization rises and the backlog queue size decreases (Figure 5.3). In the next diagram, the resource average price displays a drop when the discount is enforced (Figure 5.4). In the auction model (Figure 5.5), the proportionalities hold as in the other two models. However, the auction model manages to maintain a higher level of resource utilization, as it can match more application tasks to resources(in every round all available resources are matched). Because the auction model maintain a more

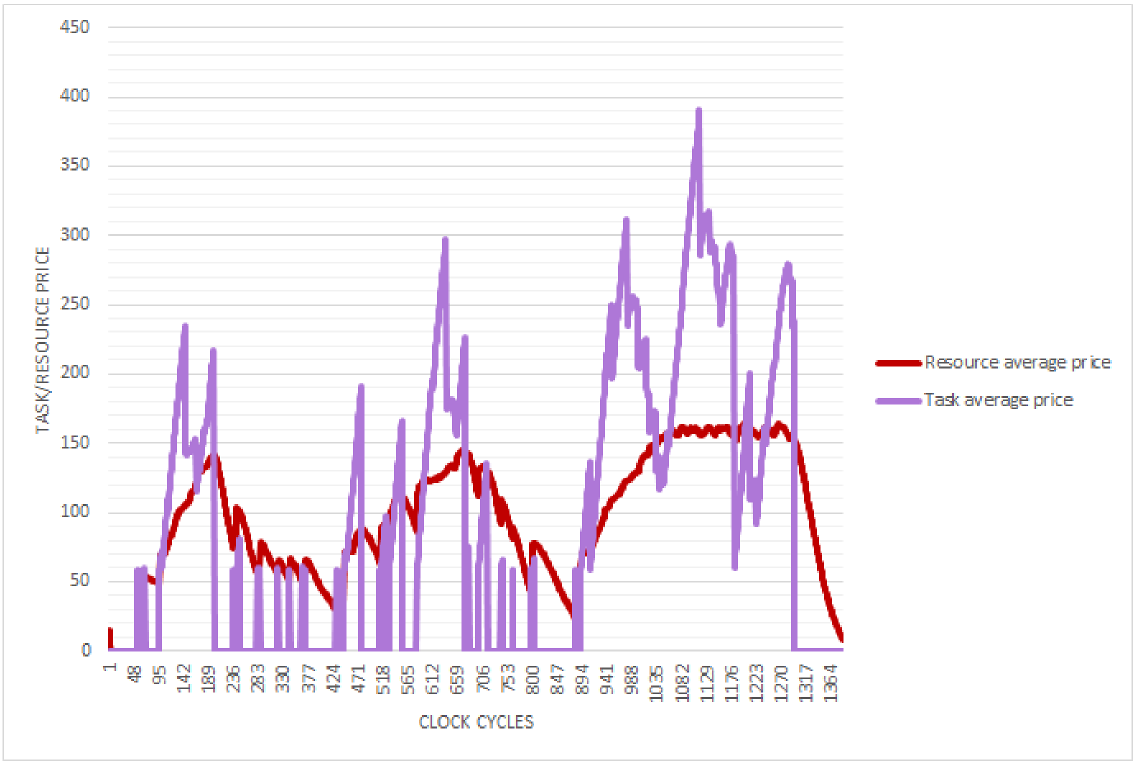

Figure 5.2: Commodity model resource and task price fluctuations over time

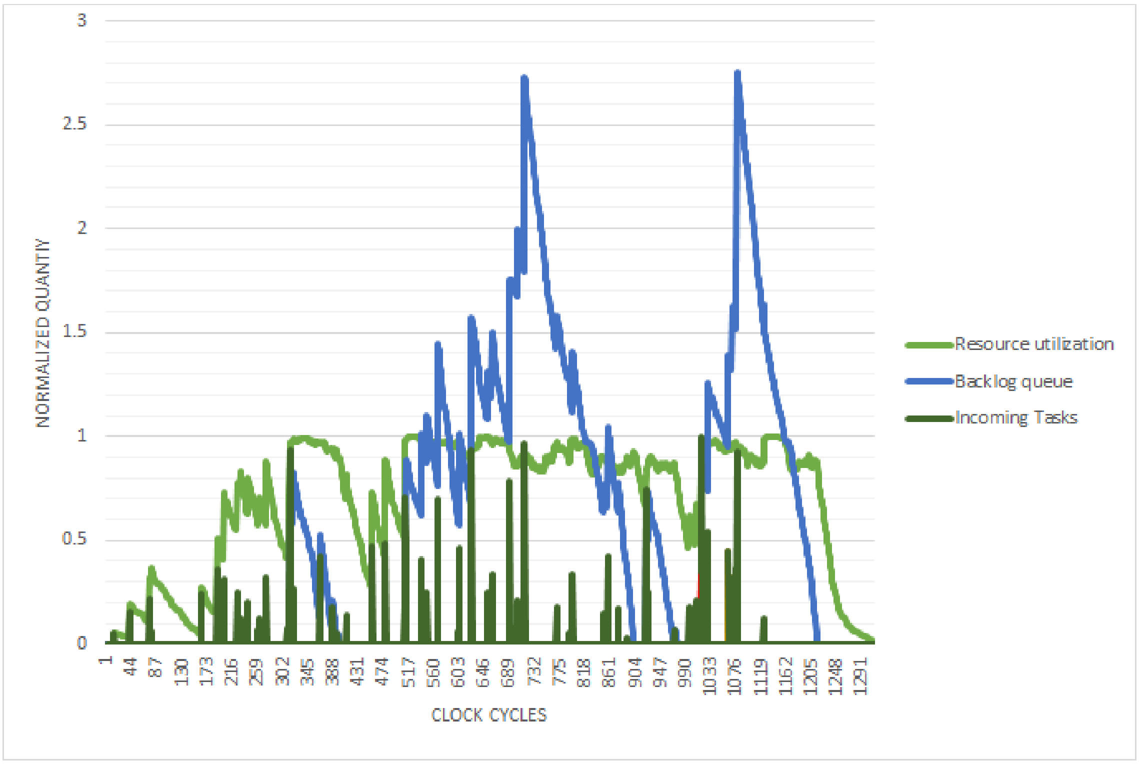

Figure 5.3: Behavior of Posted Price over time

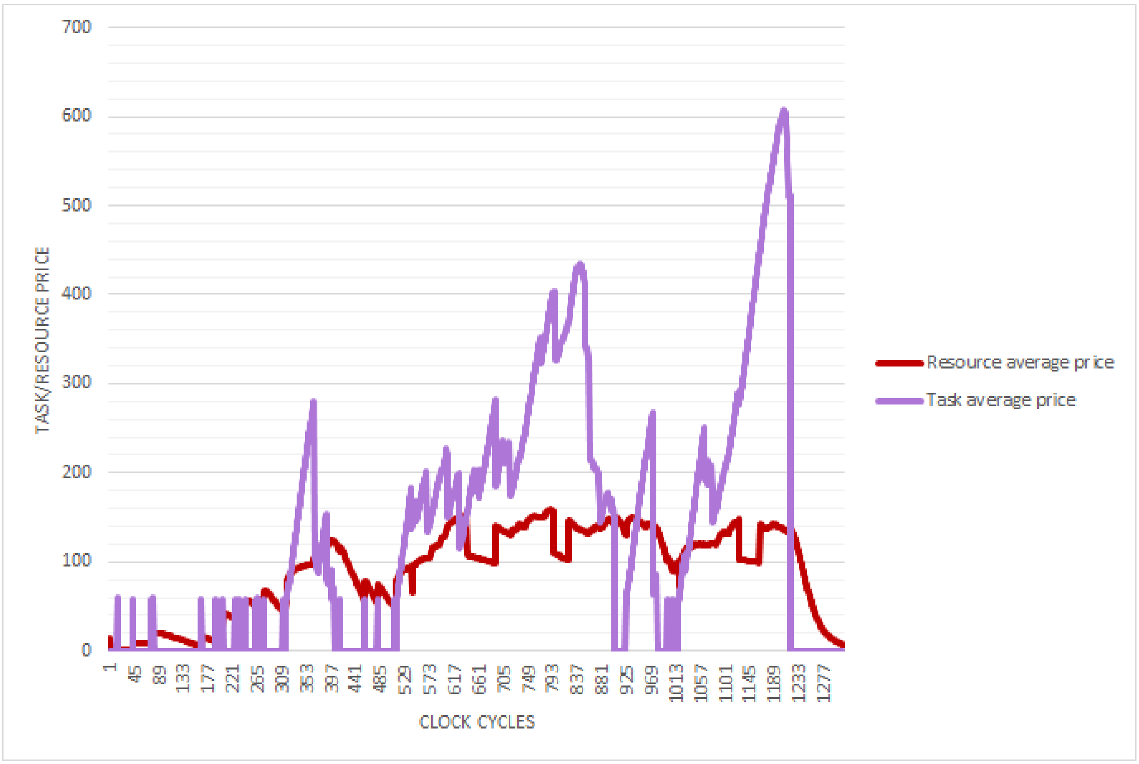

Figure 5.4: Posted Price model resource and task price fluctuations over time

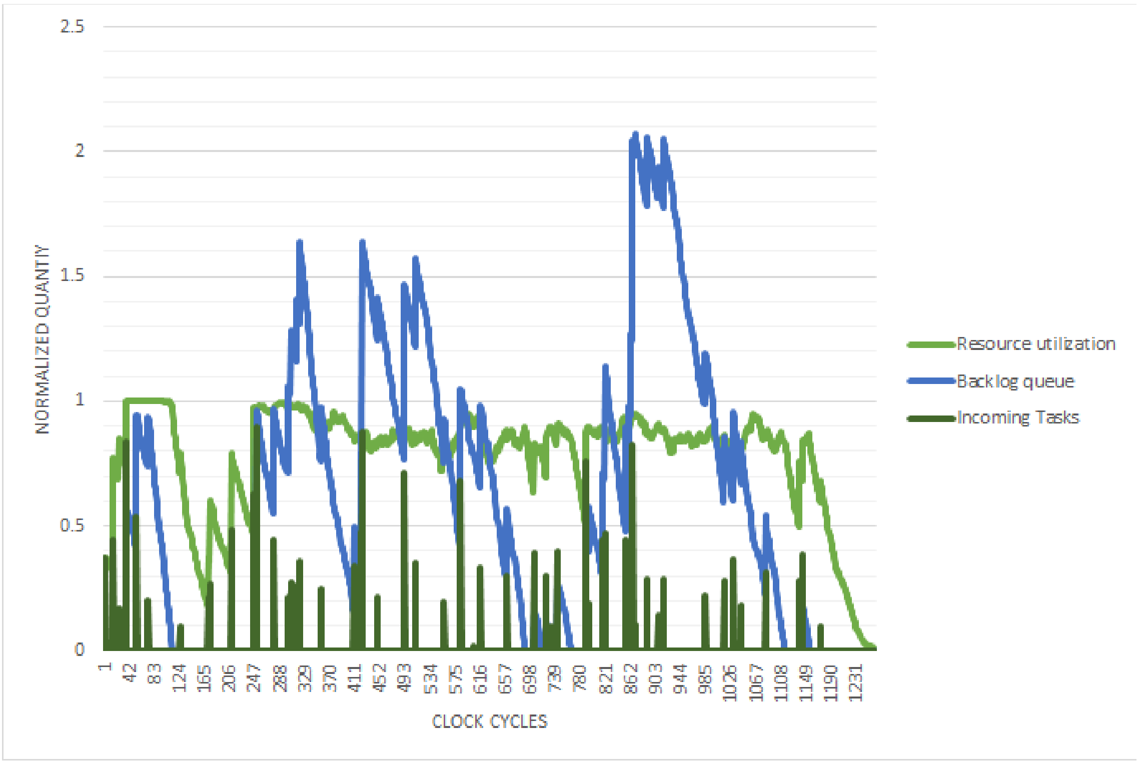

Figure 5.5: Behavior of Auction model over time

balanced average resource utilization, the average resource price is also more balanced (Figure 5.6). In Figure 5.7 the load balancing efficiency of the three different economic

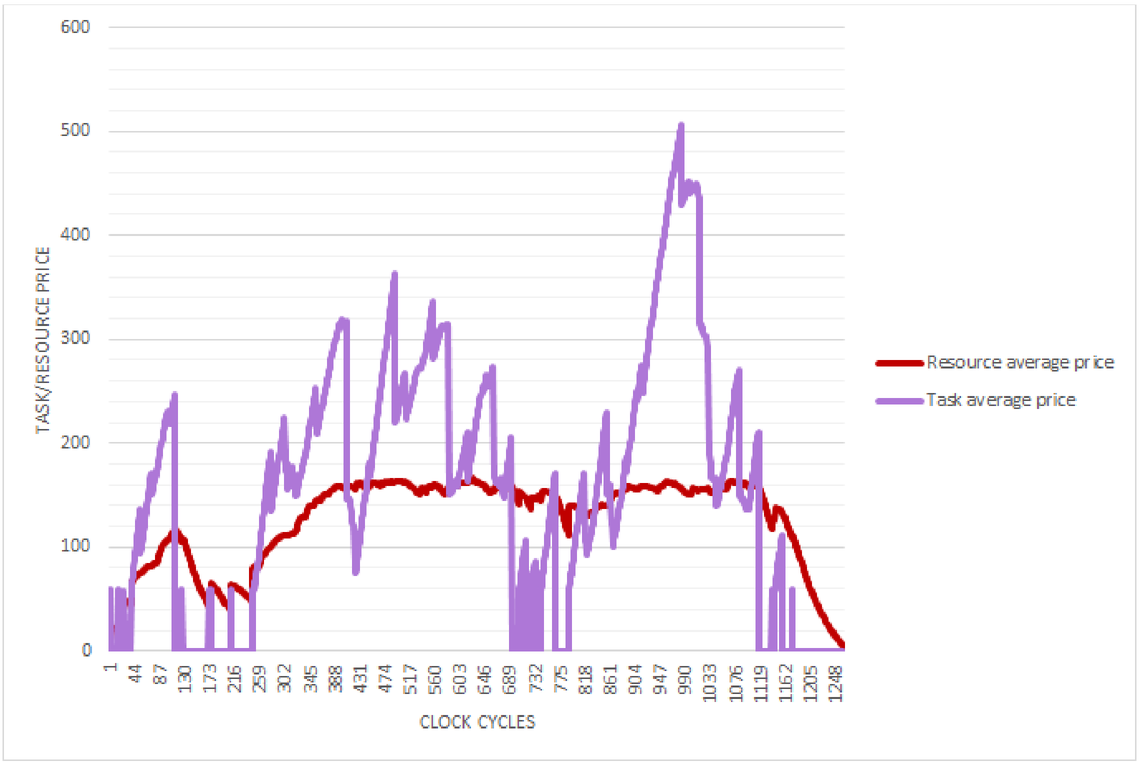

Figure 5.6: Auction model resource and task price fluctuations over time

models is illustrated. Both the Auction and the Commodity Market model perform better than the Posted Price model. The Commodity Market and Auction models perform similarly in terms of load balancing efficiency, but the Auction model performs better than the Commodity Market model in terms of execution time performance. In Figure 5.8, we observe the average resource utilization for the different resource configurations. In Figure 5.9, we can observe the different execution times for the different resource configurations. As the number of resources increases exponentially, the execution time also decreases exponentially, as the computational power doubles as well. In Figure 5.10, we can observe the difference in execution time performance between the Commodity Market model and the Auction model. The Commodity Market model performance is used as a reference and therefore is at 1 in all configurations. The Auction model illustrates improved performance and therefore reduced execution time. Finally, in Figure 5.11 we can see the same type of diagram between all three different models.

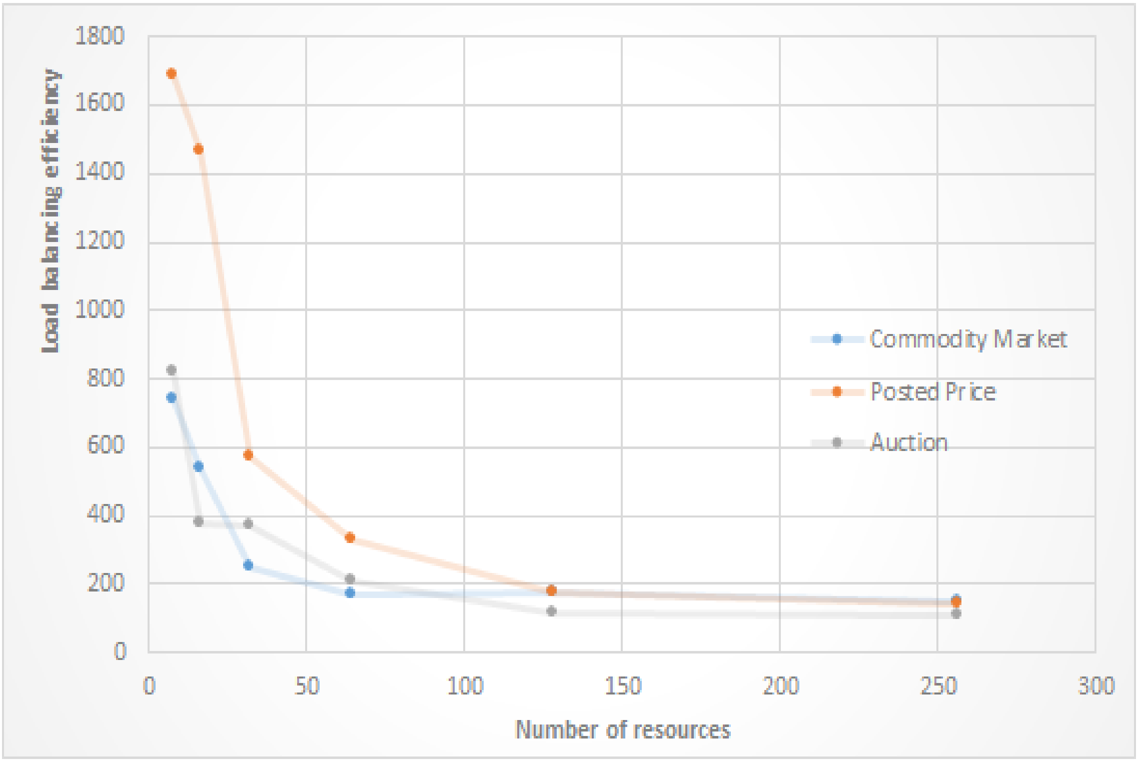

Figure 5.7: Comparison of load balancing efficiency

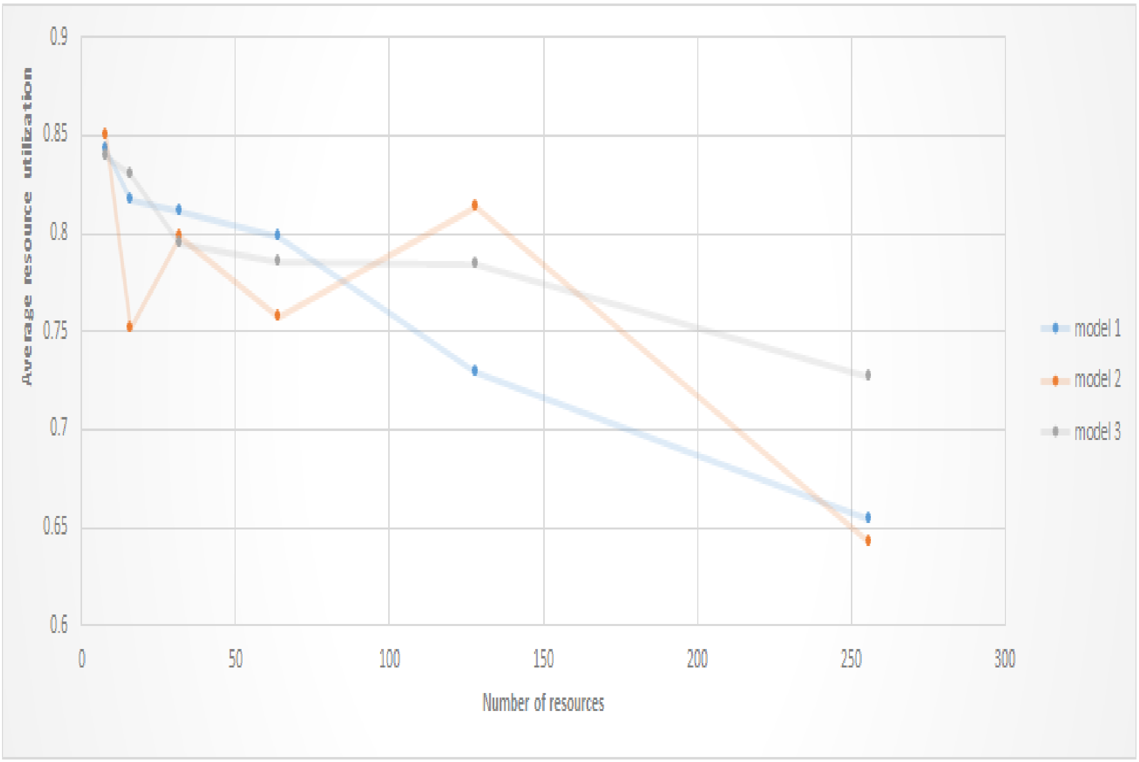

Figure 5.8: Comparison of average resource utilization

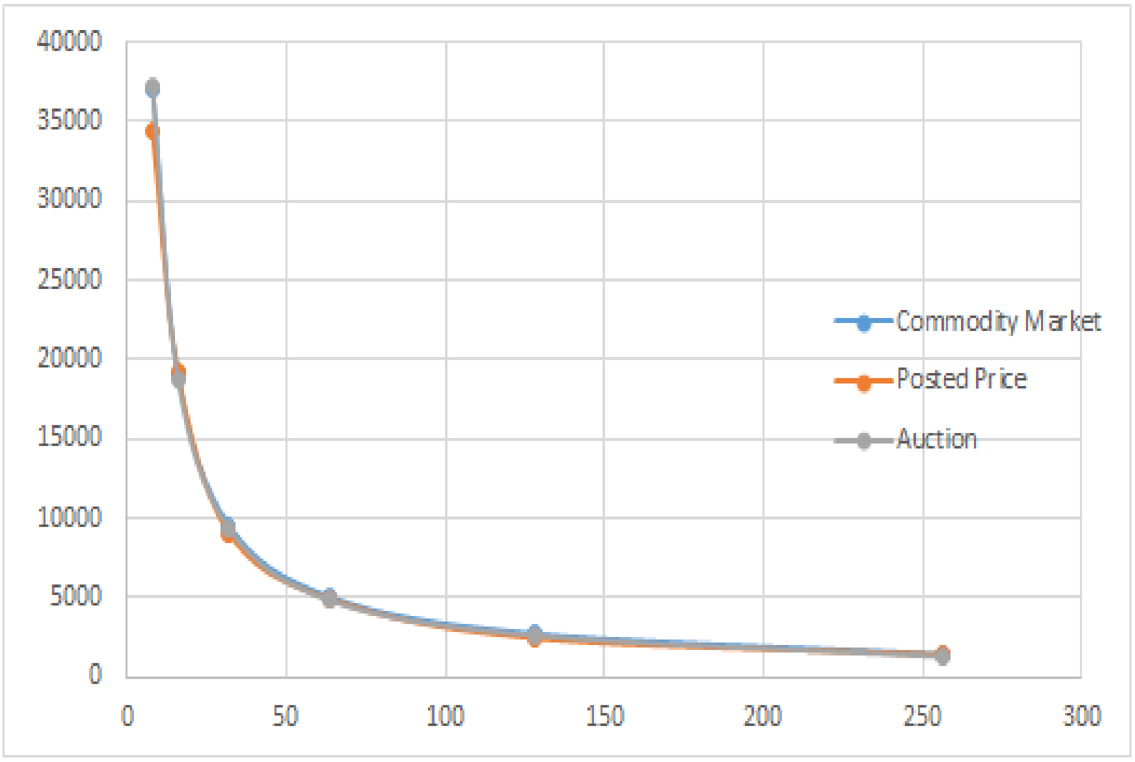

Figure 5.9: Comparison of performance between all models

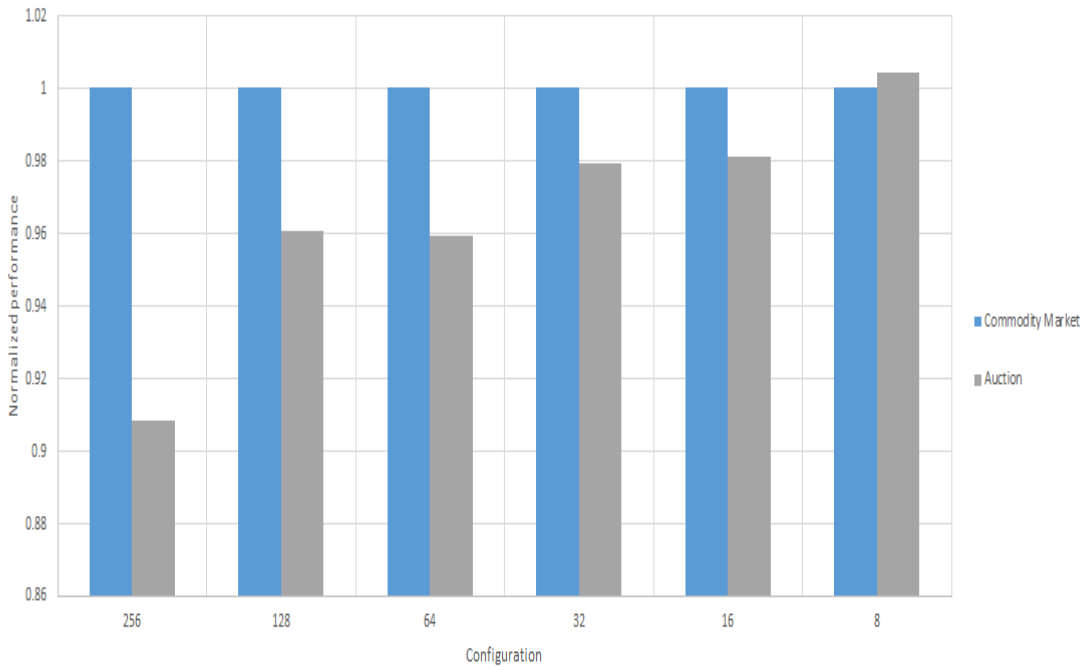

Figure 5.10: Comparison of performance between the Commodity Market and Auction models

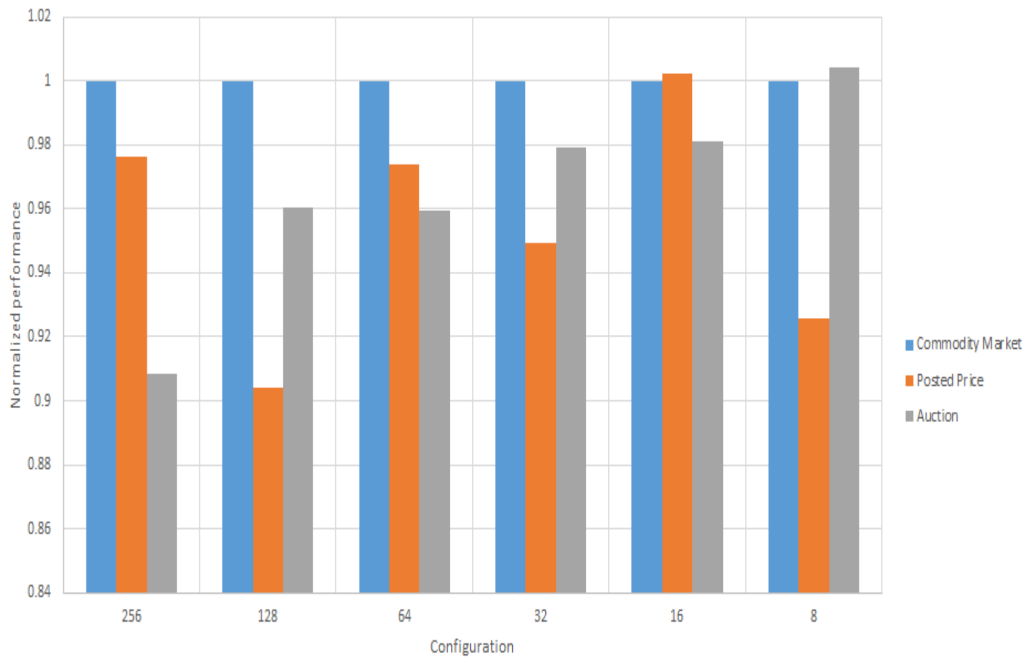

Figure 5.11: Comparison of performance between all models

Chapter 6

Conclusion and Future Work

6.1 Summary

In the current thesis, we were involved with the problem of distributed run-time resource management for Many-core systems, based on economic principles. Initially, an introduction is presented, giving an overview of Many-core Processor Architectures and of scheduling and mapping in this type of platforms. Additionally, background knowledge regarding basic economic principles is presented. The definition of the problem and its contribution is discussed after the study of related work in the field.
The proposed methodology targets homogeneous Many-core systems and provides them with a three market-based models that handle the performance-reliability tradeoff in distinct ways. Results show the different behavior of each model and their efficiency in making reliability aware decisions is discussed and compared.

6.2 Future Work

In the following sections, there are proposed ways of enhancing the proposed methodology by either incorporating extra features or by testing and evaluating it in different settings.

6.2.1 Intel SCC and Intel Xeon Phi port

The models developed have been implemented at a simulation level. A real Many-core platform could leverage the proposed methodology and indicate some bottlenecks as well. Both the Intel SCC and the Xeon Phi could provide a real platform for the methodology to test its scalability and to perform in a multiresource and highly parallel environment. By porting the framework, an opportunity to create and test distributed computation mechanisms would be given.
Regarding reliability, the Intel SCC provides support for Dynamic Voltage and Frequency Scaling. By being able to adjust these core parameters, we could extend our methodology to translate market decisions into voltage and frequency settings.

6.2.2 Task Migration

Task migration is the process of tranferrring a running application from a core to another. It can seriously affect any resource management approach. An implementation of a task migration mechanism can help achieving better results, as matching choices can become fine grained. A task migration mechanism should be carefully considered, as in a centralized mapping architecture as the one implemented, it could cause significant overhead for the entire platform. A bottleneck could arise by migrating memory-bound applications.
In terms of improvement, task migration could significantly improve load balancing efficiency as well as reliability. By making fine grained choices, the effect of the error

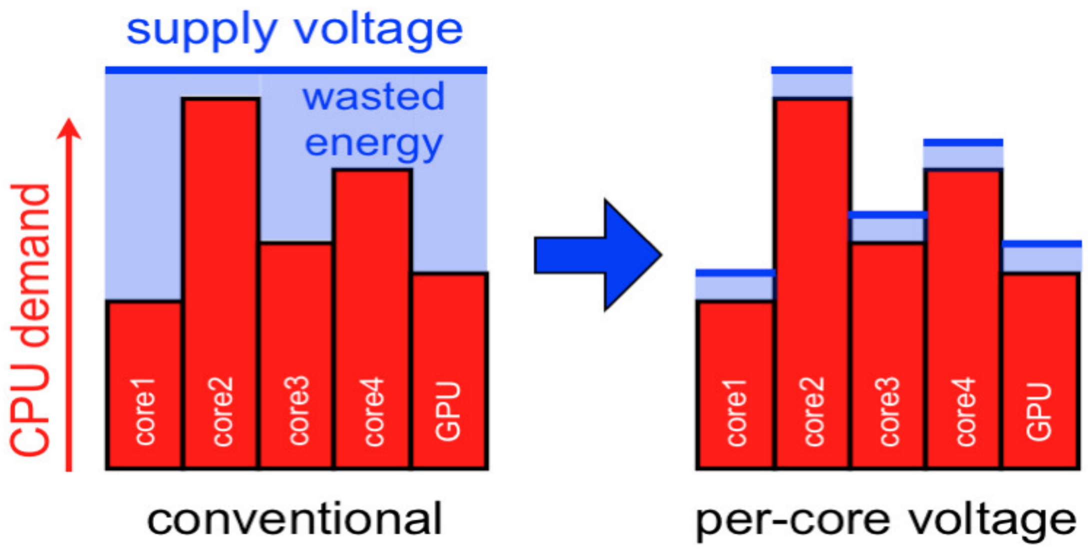

Figure 6.1: Using DVFS to aim at reliability

introduced by false predictions could be ameliorated. In general, a better resource utilization can be achieved. Finally, excessive use of specific cores can be avoided in order to improve reliability.

6.2.3 Parsec Benchmarks and Input Distributions

Currently, the workload used in the framework is generated at runtime. In order for the framework to become more realistic, real-world workload could be used. Benchmark suites, that feature modern parallel workload are available (e.g. PARSEC benchmark suite [5]). Using a benchmark suite would demand better prediction mechanisms, but would create results that display a behavior closer to real Many-core systems.
Apart from the benchmark suite, statistical distributions could be used to define the behavior of the incoming application tasks to the resource management framework. These distributions would define how many application tasks would enter the resource manager at a specific clock cycle. An example of a Poisson distribution is illustrated in Figure 6.3.

6.2.4 Implementing a model for distributed mapping

The methodology developed is based on centralized mapping. A master node is responsible for all communication and for managing all other cores. A distributed matching approach would be interesting, as it would reduce the communication overhead with the master node. It could also provide the basis for the development of another model. Such a model would need to partition the available resources and assign managers to every partition. In economic terms, this could be market consisting of sub-markets. The manager of the partition resembles to an agent that coordinates the market between buyers and

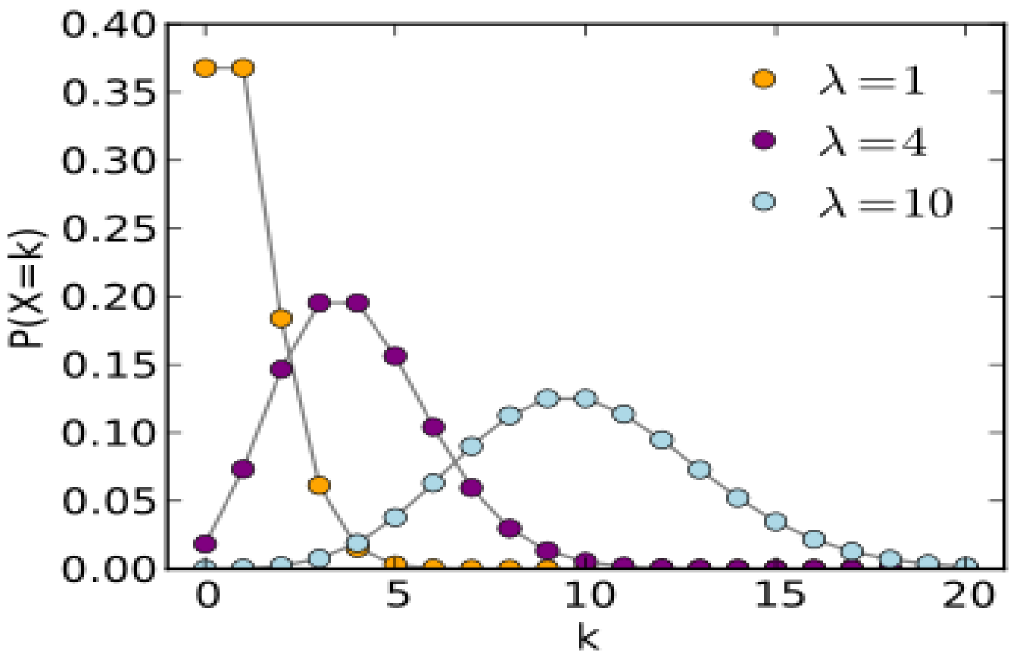

Figure 6.2: A Poisson Distribution [34]

sellers. By using this approach, we could evaluate distributed mapping using agent-based markets.

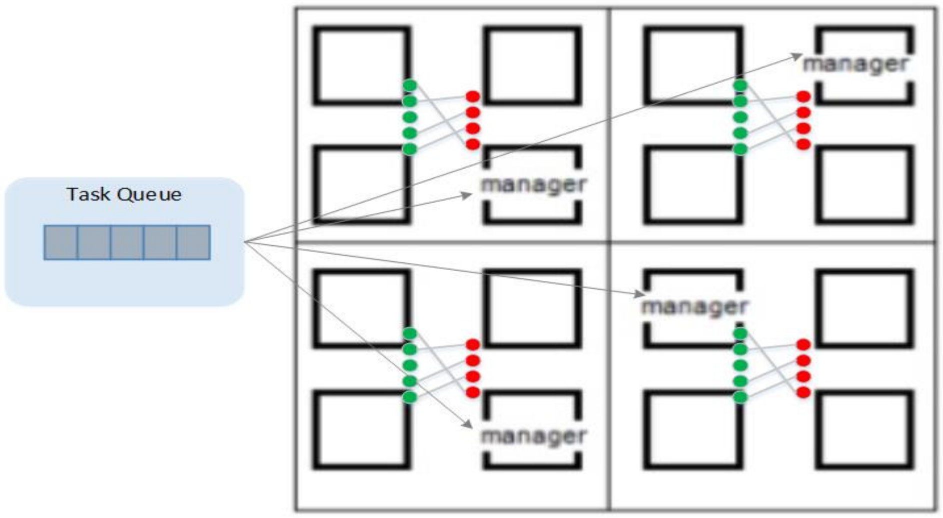

Figure 6.3: Agent-based Market model for distributed mapping

Bibliography

[1] Mridul Agarwal, Bipul C. Paul, Ming Zhang and Subhasish Mitra. Circuit failure prediction and its application to transistor aging. Στο *Proceedings of the 25th IEEE VLSI Test Symmposium*, VTS '07, σελίδες 277–286, Washington, DC, USA, 2007. IEEE Computer Society.

[2] Mohammad A. Al Faruque, Rudolf Krist and Jörg Henkel. ADAM: run-time agent-based distributed application mapping for on-chip communication. Στο *DAC '08: Proceedings of the 45th annual Design Automation Conference*, σελίδες 760–765, New York, NY, USA, 2008. ACM.

[3] Iraklis Anagnostopoulos, Alexandros Bartzas, Georgios Kathareios and Dimitrios Soudris. A divide and conquer based distributed run-time mapping methodology for many-core platforms. Sto Rosenstiel kai Thiele [3], σελίδες 111–116.

[4] Iraklis Anagnostopoulos, Alexandros Bartzas, Georgios Kathareios and Dimitrios Soudris. A divide and conquer based distributed run-time mapping methodology for many-core platforms. Στο *DATE*Wolfgang Rosenstiel and Lothar Thiele, επιμελητές, σελίδες 111–116. IEEE, 2012.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Στο *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[6] Leo Borges. An Introduction to the Intel Xeon Phi coprocessor, 2013.

[7] Léon Bottou. Stochastic learning. Στο *Advanced Lectures on Machine Learning*Olivier Bousquet and Ulrikevon Luxburg, επιμελητές, Lecture Notes in Artificial Intelligence, LNAI 3176, σελίδες 146–168. Springer Verlag, Berlin, 2004.

[8] Rajkumar Buyya, David Abramson, Jonathan Giddy and Heinz Stockinger. Economic models for resource management and scheduling in grid computing. σελίδες 1507–1542. Wiley Press, 2002.

[9] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović

and John D. Kubiatowicz. Tessellation: Refactoring the os around explicit resource containers with continuous adaptation. Στο *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, σελίδες 76:1–76:10, New York, NY, USA, 2013. ACM.

[10] Altera Corporation. From Multicore to Many-Core: Architectures and Lessons. `http://www.altera.com/technology/system-design/articles/2012/multicore-many-core.html`, 2012.

[11] Intel Corporation. Intel Xeon Phi Coprocessor Datasheet, 2014.

[12] Tilera Corporation. Tile Processor Architecture overview for the TILEPro Series. `http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf`, 2013.

[13] Ayse K. Coskun, Tajana Simunic Rosing, Yusuf Leblebici and Giovanni De Micheli. A simulation methodology for reliability analysis in multi-core socs. Στο *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, GLSVLSI '06, σελίδες 95–99, New York, NY, USA, 2006. ACM.

[14] Gregory Frederick Diamos. *HARMONY: An Execution Model for Heterogeneous Systems.* Ph.D. dissertation, 2011.

[15] Allen B. Downey. A model for speedup of parallel programs. Τεχνική Αναφορά υπ. αρίθμ., 1997.

[16] Embedded Systems Laboratory (ESL) EPFL. 3D Stacked Architectures with Interlayer Cooling (cmosaic), 2012.

[17] Mohammad Abdullah Al Faruque, Rudolf Krist and J?rg Henkel. Adam: run-time agent-based distributed application mapping for on-chip communication. Στο *DAC*Limor Fix, επιμελητής, σελίδες 760–765. ACM, 2008.

[18] Joseph A. Herriges. Chapter 3: Supply and Demand: A Model of a Competitive Market. `http://www2.econ.iastate.edu/classes/econ101/herriges/Lectures10/Chapter%203H%20-%20Supply%20and%20Demand.pdf`, 2010.

[19] Lee Hock Koon, Sanjeev Solanki and Xie. Yilin. A Day at the STO. `http://www.dsi.a-star.edu.sg/e-newsletter/Oct2012/STO.html`, 2012.

[20] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. Στο *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, σελίδες 233–239, New York, NY, USA, 2003. ACM.

[21] Microsoft Parallel Computing Initiative. The manycore shift white paper, 2007.

[22] TheInternational Technology Roadmap for Semiconductors. International technology roadmap for semiconductors 2009 edition, 2009.

[23] D.A. Patterson. J.L. Hennessy. *Computer Architecture: A Quantitative Approach,*. Morgan Kaufmann, 2007.

[24] Wonyoung Kim. Fast, per-core dvfs using fully integrated voltage regulators. `http://www.eecs.harvard.edu/~wonyoung/research.html`, q.q.

[25] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schr?der-Preikschat and J?rg Henkel. Distrm: distributed resource management for on-chip many-core systems. Sto Dick kai Madsen [25], σελίδες 119–128.

[26] Takashi Kunimoto. Lecture Note on Auctions. Department of Economics, McGill University, 2008.

[27] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl and Saurabh Dighe. The 48-core scc processor: The programmer's view. Στο *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, σελίδες 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[28] Andrew Ng. Chapter 2: Linear Regression With One Variable. `https://d396qusza40orc.cloudfront.net/ml/docs/slides/Lecture2.pdf`, 2014.

[29] Jochem H. Rutgers. *Programming Models for Many-Core Architectures, A Co-design Approach.* Ph.D. dissertation, 2014.

[30] Βασίλειος Σ. Τσούτσουρας. *Design And Implementation Of A Run-time Resource Manager For Malleable Applications On Network-on-chip (noc) Architecture.* Διπλωματική Εργασία, 2013.

[31] Ηρακλής Αναγνωστόπουλος. *Μεθοδολογίες και εργαλεία διαχείρισης πόρων και παραμετροποίησης εφαρμογών κατά το χρόνο εκτέλεσης σε πολυπύρηνες ενσωματωμένες πλατφόρμες.* Διδακτορική Διατριβή, 2014.

[32] Mattson Tim and Robvan der Wijngaart. RCCE: A small library for many-core communication, 2010.

[33] András Vajda. *Multi-core and Many-core Processor Architectures.* Springer US, 2011.

[34] Wikipedia. Poisson distribution. `http://en.wikipedia.org/wiki/Poisson_distribution`, q.q.

[35] Chee Shin Yeo and Rajkumar Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Softw. Pract. Exper.*, 36(13):1381–1419, 2006.

[36] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, 2012.