ulm university universität
uulm

Faculty of Engineering, Computer Science and Psychology
Institute of Information Resource Management

# A Performance Analysis Model for a Management Architecture of Multicore Processing Units

Master's Thesis

by

Stefan Bonfert

01.11.2015

Supervisor: Dipl.-Inf. Vladimir Nikolov
Co-supervisor: Dipl.-Inf. Lutz Schubert
Examiner: Prof. Dr.-Ing. Stefan Wesner
Co-examiner: Prof. Dr.-Ing. Franz J. Hauck

ulm university universität
**u**ulm

Faculty of Engineering, Computer Science and Psychology
Institute of Information Resource Management

Master's Thesis

# A Performance Analysis Model for a Management Architecture of Multicore Processing Units

## Abstract

In the field of high performance computing the partitioning of a problem into smaller problems, that then are solved in parallel on multiple computing units, is a crucial task. In current software architectures the solving of those smaller problems on the operating system and application layer is carried out by threads, which have to be created and managed by the operating system. These tasks are computationally expensive. Because of the monolithic kernel of most modern operating systems, the tasks of managing and activating threads are sequentially performed by a central instance and are therefore hard to parallelize. For high performance systems, utilizing a large number of processing units, this centralized processing becomes a bottleneck. Operating system architectures, that are tailored for manycore processors, aim for parallelization and distribution of internal tasks of the kernel. This also applies to the management of threads, for which currently no universal model exists. Although a performance gain can be expected when using a distributed management of threads, additional tasks, like the synchronization of state or the delegation of tasks between the management instances, may impact the performance.

In this thesis a generic model for a distributed thread management is developed. The influence of the number of thread management instances on the performance of applications is examined. This model offers a possibility to derive an optimal configuration of management instances in respect of distribution and communication overhead for a specific hardware configuration. This configuration significantly influences the latencies for creation of different amounts of threads, which should be minimized. The developed model is evaluated and validated by deriving a description for a specific hardware configuration. This description is then realized by an appropriate thread management facility which is implemented and integrated within a specialized OS.

# Kurzfassung

Im Bereich des High Performance Computing besteht eine entscheidende Aufgabe darin, das zu lösende Problem in Teilprobleme zu zerlegen und diese möglichst asynchron und parallel auf unterschiedlichen Recheneinheiten zu lösen. In gängigen Softwarearchitekturen werden diese Teilaufgaben auf der Anwendungs- und Betriebssystemebene von Threads ausgeführt, deren Erzeugung und Verwaltung aufwändig ist. Da die Mehrheit heutiger Betriebssysteme auf einen monolithischen Kernel aufsetzt, sind die Aufgaben zur Verwaltung und Aktivierung von Threads meist sequentiell bzw. nur schwer parallelisierbar und werden zentral ausgeführt. Für High Performance Systeme, die mit einer großen Anzahl von Recheneinheiten arbeiten, stellt diese zentrale Verarbeitung einen Flaschenhals dar. Betriebssystemarchitekturen, die auf Vielkern-Prozessoren spezialisiert sind, zielen darauf ab, kernelinterne Aufgaben möglichst zu parallelisieren und zu verteilen. Dies gilt ebenfalls für die Verwaltung von Threads, für die bislang kein allgemeingültiges Modell existiert. Obwohl eine verteilte Verwaltung von Threads einen Performanzvorteil verspricht, fallen dabei zusätzliche Aufgaben an, wie z.B. die Synchronisation des Zustands oder die Delegierung von Aufgaben zwischen verschiedenen Verwaltungsinstanzen.

In dieser Arbeit wird ein generisches Modell für eine verteilte Threadverwaltung erstellt. Es wird untersucht, welchen Einfluss die Anzahl der verwendeten Verwaltungsinstanzen auf die Performanz von Anwendungsprogrammen hat und wie dieser Einfluss beschrieben werden kann. Anhand dieses Modells ist es möglich, für eine spezielle Hardware-Konfiguration eine möglichst optimale Konfiguration der Verwaltung hinsichtlich ihrer Verteilung und ihres Synchronisationsoverheads zu bestimmen. Diese Konfiguration beeinflusst maßgeblich die Latenz bei der Erzeugung unterschiedlicher Thread-Mengen, die minimiert werden soll. Das erarbeitete Modell wird anhand einer damit extrahierten optimalen Beschreibung und Implementierung der Threadverwaltung für eine konkrete Hardware-Konfiguration evaluiert und validiert.

---

| | |
|---|---|
| Date of issue: | 01.05.2015 |
| Date of submission: | 01.11.2015 |
| Author: | Stefan Bonfert |
| Supervisor: | Dipl.-Inf. Vladimir Nikolov |
| Co-supervisor: | Dipl.-Inf. Lutz Schubert |
| Examiner: | Prof. Dr.-Ing. Stefan Wesner |
| Co-examiner: | Prof. Dr.-Ing. Franz J. Hauck |

I hereby declare that this thesis titled:

**A Performance Analysis Model for a Management Architecture of Multicore Processing Units**

is the product of my own independent work and that I have used no other sources and materials than those specified. The passages taken from other works, either verbatim or paraphrased in the spirit of the original quote, are identified in each individual case by indicating the source.
I further declare that all my academic work has been written in line with the principles of proper academic research according to the official "Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis" (University Statute for the Safeguarding of Proper Academic Practice).

Ulm, 01.11.2015

Stefan Bonfert

# Contents

# 1 Introduction

Ever since the introduction of multicore processors, developers are facing a new challenge. They have to exploit parallelism in their programs in order to gain additional performance. To solve this challenge, they are supported by both hardware and software. Modern processors contain multiple cores, each consisting of multiple hardware threads. They are able to run code in a parallel or quasi-parallel manner. Modern operating systems offer the concept of threads to manage the parts of the code that are executed in parallel.

The number of processing units managed by a single operating system increased in recent years and will continue following this trend. In the past, it was necessary to assign multiple threads to a single processing unit and switch between these threads. This time multiplexing is accomplished by an appropriate scheduler. In future systems with large amounts of processing units it may be feasible to assign a single thread to each one, since the number of processing units massively exceeds the number of applications. Therefore, no context switches may be necessary, eliminating performance impacts from scheduler operations, cache pollution and TLB invalidations.

In most operating systems, resources, including processing units, are managed in a centralized way. Shared data structures are used to track their occupation status. This management architecture is not scalable to recent and future systems with large amounts of processing units. The central management of resources causes a large overhead for synchronization, that is required when accessing shared data structures.

A decentralized and distributed architecture for the management of a system's computational resources is required in order to reduce the overhead of thread creation. For current applications it may be possible to offer increased performance and a higher degree of parallelism due to the reduced overhead. A decentralized management architecture may still require synchronization operations, that impact the performance. Since the frequency of these operations depends on the degree of decentralization, the performance of the management architecture degrades, if many management instances are utilized. On the other hand, performance is not optimal, if too few management instances are used. Therefore, an optimal degree of decentralization

exists. However, up to this point there is no possibility to determine this optimum for the decentralized management of computational units.

In this thesis a distributed thread management facility for high performance computing (HPC) systems, which are introduced in chapter 3, is developed. Its task is to manage the occupancy status of individual processing units inside a single manycore processor, that runs a single application. A performance model is created for the specific case of thread creation and the management of computational resources, which describes the operation of this thread management facility. This model yields the possibility to determine the optimal degree of decentralization of thread management for a particular application and an underlying computing platform. The model considers both the overhead of thread creation and the communication among different management instances. In order to determine the optimal degree of decentralization, the model calculates the response time for requests by the application for different amounts of management instances. Appropriate characteristics of both hardware platform and application are used as parameters for the performance model to make it applicable to arbitrary platforms and workloads. The performance model and its parameters are detailed in chapter 4.
Additionally, an implementation of the developed distributed thread management facility is conducted. To allow individual management instances to operate independently from each other, their computations are performed on private data structures. Therefore, no synchronization is required when accessing these data structures. Asynchronous messages are exchanged between the different management instances, where necessary. The distributed thread management facility developed in this thesis shall improve the performance of requesting applications by parallelizing the thread creation overhead over a set of management instances. While the mode of operation of the developed management facility is described in chapter 4, its implementation and internal operation are presented in chapter 6.
The implementation of the thread management facility is benchmarked using different sample applications, that are running on an evaluation platform introduced in chapter 5. The results of this benchmark are compared to theoretical values obtained from the performance model. Thereby, the performance model is validated and evaluated. The theoretical and experimental response times are compared, as well as the optimal degree of decentralization. The results of this process are presented in chapter 7.

# 2 Related Work

This thesis focuses on the management of computational units inside a HPC system. In this chapter some recent approaches to the management of those inside an operating system are presented. In order to provide a scalable and performant infrastructure, the management architecture developed in this thesis needs to be decentralized. Therefore, approaches that consider decentralized management schemes are covered in section 2.1. In recent years a new field appeared, which is called invasive computing. A computer's resources are managed by a thin layer, that provides fast access to those. Since this approach promises to optimize an application's performance, it is covered in section 2.2.

## 2.1 Decentralized Management

Decentralized resource management is required to provide performant systems for growing amounts of processor cores. The authors of [FKH08] propose a decentralized infrastructure to map tasks to individual processor cores. For this purpose the chip is segmented into virtual clusters, each managed by a cluster agent. An algorithm to select a cluster capable of running the tasks of a task graph onto processing elements is presented. This algorithm utilizes information about energy consumption per processing element type, resource requirements by the tasks and availability information. If no suitable cluster can be found, task migration and reclustering are used to rearrange tasks and clusters in order to create a suitable cluster. After a cluster is selected to run the tasks, a heuristic algorithm is employed to map tasks to individual processing elements, that considers communication costs and computation costs per processing element type.
A similar approach was chosen in [ATBS13], where each application's resources are managed by a manager core, which is also responsible for optimizing the application's performance. The performance model is based on [Dow97]. Manager cores are allowed to trade worker cores in order to achieve maximum performance for their applications.
While these strategies cause considerable overhead for communication during setup

and reclustering, the management scheme presented in [ZSU+09] relies on a heuristic approach. A task graph is used to map the tasks to individual processing elements for execution. A heuristic algorithm is then used to improve this mapping to optimize performance.

These management strategies have two drawbacks. First, they require detailed knowledge about the internals of the application and the utilized processors. Task graphs and the task's runtimes on the processors have to be extracted from the application using a compiler integration. Second, these approaches take away control from the application developer, since the developer can not influence the partitioning of the computational resources.
Therefore, the management architecture developed in this thesis gives developers direct control over the resources, while the developed performance model uses parameters of both hardware platform and application that are easily accessible.

The allocation of CPU cores was also covered in [KBL11]. In this paper, an algorithm for the allocation of regions of cores, i.e. cores close to each other, is given, where the information about the occupancy state of the cores is stored in a distributed directory. The amount of communication is reduced by communicating with single cores, that then collect information about their local neighborhood and send an aggregated response. A task is migrated to another core to create larger regions for other applications, if it only suffers a small delay thereby. After the setup of the regions, cores are only migrated if a core has a much higher gain, than the other core's loss. Using this method, ensuring allocation of cores close to each other for individual applications, they were able to reduce the communication traffic by up to 83% compared to centralized management.

All of these decentralized management schemes are based on initial assumptions about the number of management instances. Either a fixed number of instances or one instance per application is used, neglecting the utilization of these instances. An application that issues a requests to acquire new computing cores at a high rate may experience increased performance, if additional management instances are added. This case is neglected, since a fixed amount of instances is chosen.
Therefore, a performance model, like the one developed in this thesis, is required. This performance model yields the optimal degree of decentralization for the management architecture.

In [HRY+08] an interesting idea is presented, that is used for the Corey system. According to the authors, in most operating systems a reasonable amount of time is spent on system calls. If a core processes a system call, it has to acquire locks and fetch cache lines from the last core that executed a system call. This leads to long transfer times and polluted caches due to replication of data. Therefore, the authors

propose a model where kernel code is only executed by so called *kernel cores*, that are dedicated completely to only run kernel code. Other cores' system calls are then processed by these kernel cores.
In this thesis, the basic concept of dedicating cores to a single task was adopted in chapter 4.

## 2.2 Invasive Computing

In recent years, the field of invasive computing [THH+11] emerged. The main concept of invasive computing is to bring the application programs closer to the hardware by making them resource aware. This is accomplished by offering the program two operation `invade` and `retreat` to either acquire additional computational resources from the operating system or to hand them back. These operations are intended to be used to increase the level of parallelism of a program and to make resources available again, when they are not required anymore. According to the authors, the operating system should offer some interfaces to determine, if invading additional resources is reasonable, e.g. interfaces for reading temperatures or the current load.
This idea was implemented by OctoPOS [OSK+11], among others, which defines a claim as a set of available cores, that can be characterized in different ways like coherence, homogeneity and spatial and temporal preemptibility. The authors provide implementations of methods for `invade` and `retreat` for every combination of these characterizations. They are therefore highly specialized to achieve high performance for every claim. However, even for this simple characterization, they have to provide 16 implementations of the same method.

To overcome the limitations of these systems, a decentralized management architecture for computational resources is developed in this thesis. To determine the optimal amount of management instances, a performance model to determine a performance measure is developed. This model only uses information about the platform and application programs that is easily acquirable, e.g. the bandwidth available for communication. This allows software developers to incorporate the proposed scheme with little additional effort.

# 3 Technical Background

In this chapter, the technical principles this thesis is based on are introduced. A performance model for managing computational units in HPC is developed. The aim of this management facility is to remove the bottleneck of cenalized management. Important properties of HPC systems that influenced design decisions made in this thesis, are described in section 3.1. To validate the developed performance model an implementation of a management architecture for hardware threads (HWTs) was evaluated on a suitable platform, which is based on a microkernel, whose concept is detailed in section 3.2.

## 3.1 High Performance Computing

The field of high performance computing (HPC) deals with workloads that require more computing power, than a normal desktop or server computer offers. Often, these workloads are generated by simulations in both science and industry [HW10]. Since these simulations get increasingly complex and model reality more detailed, they require increased processing power in order to finish their calculations in a reasonable time.
The processing of these HPC workloads requires the utilization of multiple central processing units (CPUs), that share the workload. While a few CPUs can be put into the same enclosure, many applications require the combined computing power of multiple enclosures [SD11]. These are connected using an high-bandwidth, low-latency interconnect and form a coherent system. The fastest of these systems are regularly benchmarked and the 500 most performant systems are collected into the TOP500 list [MSD+15]. Figure 3.1 shows the average amount of CPU cores per system for all systems in the TOP500 list over the years. A dramatic increase in the amount of CPU cores can be observed.
To take advantage of the increasing performance these systems offer, application developers have to exploit parallelism, that is inherent to their programs. This is accomplished by creating separate threads, which are executed by the operating system on the available CPUs. It is the operating system's task to assign these threads to

individual CPU cores or hardware threads (HWTs) for execution.

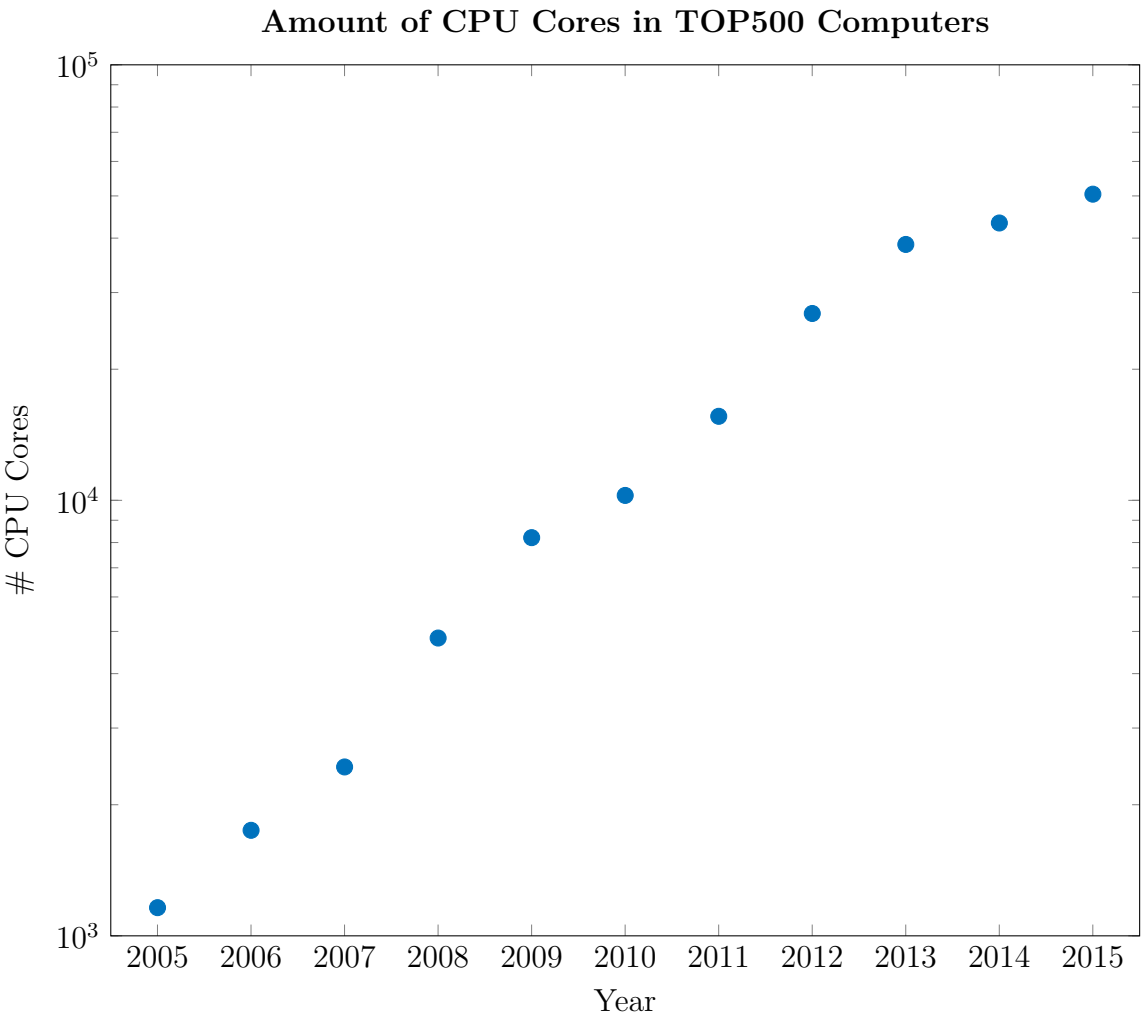



**Figure 3.1:** The average amount of CPU cores in HPC systems included in the TOP500 ranking; Source: [MSD+15]

In common server or desktop applications, the number of concurrently running applications outnumbers the amount of available CPU cores and even the amount of hardware threads. Therefore, scheduling is required for time multiplexing between these applications and their threads and even when using only single threaded applications, the CPU could be fully utilized. In HPC applications, however, usually only one or very few applications are running simultaneously, while the computing infrastructure is much larger than on common servers and includes at least several hundred CPU cores. Therefore, the amount of applications is outnumbered by the number of CPU cores, which makes it necessary to have different threads inside an

application to utilize the computing infrastructure. While multiple threads per process are a nice-to-have feature in common server and desktop applications to improve responsiveness and make programming easier, they are an absolute necessity for HPC. On the other side, scheduling is not mandatory in HPC. In desktop and server operating systems scheduling is necessary, because one CPU core has to serve multiple applications. In HPC each application can utilize multiple CPU cores on its own. Therefore, no scheduling has to be employed for these use cases and a one-to-one mapping from hardware threads to software threads can be used.

### 3.1.1 Workload

Applications that run on HPC systems are usually highly parallelized, in order to utilize the available computing infrastructure. These applications most commonly perform simulations from various scientific fields [HW10] [CRFS07]. The application areas include fluid dynamics, particle physics, material simulations, electrical circuit simulations, pharmaceutical drug design, solid state physics, economics, and many more. The specific characteristics of an application are as diverse as the application fields. A sample application is described in the following.

Airplane manufacturers have to design the wings of airplanes such that they meet two conditions. They have to be lightweight to save fuel and, at the same time, they have to be sturdy enough not to break. To simulate the behavior of wings during a flight, HPC systems are used [Cou09]. Different flight situations are simulated, like different wind gusts. For every situation two questions are relevant. First, it has to be determined, whether the wing breaks in this situation. For this purpose the stress of different parts of the wing material are simulated while the different parts interact with each other. Threads are used to calculate the effects on different wing parts in parallel. If the wing breaks, the most interesting question is, how exactly it breaks. For this purpose additional threads are created to simulate the process of the wing breaking. This process is important for the wing designers to improve the design of the airplane's wings and understand its current flaws. The second type of tasks, that simulate the breaking wing, requires a different type of thread.
These two types of threads cannot be combined into a single thread pool. To allow developers to quickly create different kinds of threads, a suitable application programming interface (API) is required.
Therefore, the thread management facility developed in this thesis creates new threads on demand and destroys them after use. Thereby maximum flexibility is offered to developers.

### 3.1.2 Architecture

HPC systems can be built in two different variants [HW10]. One type uses shared memory throughout the complete system, while the other type uses distributed memory, where each node only can access its own memory area. However, hybrid systems exist, that combine the two approaches. If distributed memory is used, the individual nodes have to communicate via messages in order to access data that is located in the memory of another node. Therefore, in this type of machine, a program, that performs its computations only in its own memory area and accesses the memory of other nodes only if absolutely necessary, delivers the best performance.
Machines utilizing shared memory are mostly configured as non-uniform memory access (NUMA) machines [HW10]. The main memory is physically distributed among the different nodes, but each node can access all memory areas. Local memory is accessed directly, remote memory areas are accessed using the interconnect. This leads to different access times for main memory, depending on the physical location of the requested area. Most NUMA systems belong to the class of cache coherent non-uniform memory access (ccNUMA) systems. Additionally to the usage of logically shared memory, this class of systems offers cache coherency. If multiple nodes have a copy of a memory area in their local cache and this area is changed by one of the nodes, this change has to be propagated to all other nodes. Thereby, all nodes are guaranteed to access the most recent data. For this process, a cache coherency protocol is employed, e.g. the MESI protocol [SD11]. Whenever multiple nodes operate on the same data in a ccNUMA system, some time has to be spent on the cache coherency protocol. This overhead grows with increasing number of nodes that access this data.
Therefore, in shared memory systems performance can be increased, if each memory area is only used by a single node, very much like in distributed memory systems.

In traditional operating systems, data structures for the management of computational resources are accessed by all nodes. Thereby, overhead for synchronization and cache coherency is induced.
Therefore, the management architecture for computational resources developed in this thesis operates decentralized. Each of the management instances operates on private data only. If synchronization between different instances is necessary, they exchange messages, but never share any data structures. In this thesis the scope of the developed management architecture is limited to a single manycore processor. Thereby, the performance model for the management facility does not have to consider NUMA architectures, but can focus on decentralization.

## 3.2 Microkernels

Modern operating systems offer many different functionalities. They have to do memory management to assign programs with their private space in memory. For communication with the outside world, that may be a user, the internet or some other device, they process inputs to the system and emit outputs. For utilizing hardware, that is built into the computer or connected to it externally, they need to provide device drivers to efficiently and correctly address this hardware. To be able to do computations intended by a user, they need to run applications and assign computational time to them. Finally, they also need to manage themselves and the underlying hardware.
All of these tasks an operating system has to take care of can be separated into individual modules that communicate with the other modules, but do not have strong dependencies on them. When separating these modules, the operating system developer has to decide which modules should be run in kernel space and which modules of the operating system should run in user space. In traditional operating systems like Windows, Linux and BSD all operating system functionality is performed in kernel space. However, this is not necessary and may not even be a good choice in all cases. When trying to implement an operating system tailored for high availability, it may be a good choice to strictly separate the operating system into modules. If one of these modules crashes due to a software bug, the rest of the operating system can continue running, while a monolithic operating system may crash completely. Only one of the modules, the *microkernel* is running in kernel space. The other modules are put into user space to limit the damage they can deal to the system in case of a crash [Tan01]. The strict separation of the operating system into modules also yields the possibility to dynamically add and remove modules based on the current needs. In a HPC system there may be a single interactive node with the need of communicating with the user while all other nodes are worker threads, that are only used to perform calculations. These worker nodes then do not need to contain the module used for interaction with the user. This reduces the size of the kernel for those instances and therefore reduces the probability of cache misses occurring due to the kernel taking up too much space.
Due to the high customizability of individual operating system instances and due to the increased stability of microkernel based operating systems compared to traditional operating systems based on monolithic kernels, an operating system based on microkernels could be an appropriate candidate for HPC applications. Therefore, such an operating system is used as the evaluation platform for the management architecture developed in this thesis. This operating system is further detailed in section 5.2.

# 4 Performance Model

As explained in section 3.1 it is sufficient for high performance computing (HPC) applications to have one software thread per hardware thread (HWT). A management architecture that keeps track of the occupation status of processing units therefore has to manage HWTs and is called *hardware thread manager (HWTM)* in this thesis. HWTs can be requested from a management instance, which returns a handle to them. After execution of one or more tasks on the HWT, it can be returned to the manager for reuse by another part of the application. If the application requests and returns HWTs at a high rate, one instance might become a bottleneck in processing these requests. For such cases it might be better to process the requests using multiple distributed instances of the hardware thread manager (HWTM), that then are able to process requests in parallel, as they are running on different HWTs. For HPC it is important to acquire new HWTs, and thereby increase the level of parallelism, as fast as possible after issuing the request for them. In the following, the time until a request is answered is modeled based on hardware and application properties, that are introduced in section 4.1. After presenting some basic considerations in section 4.2 a performance model for one management instance is derived in section 4.3. Afterwards, this model is extended to two management instances in section 4.4 and adapted to consider an arbitrary number of management instances in section 4.5. This performance model allows developers to calculate the number of management instances that delivers the best response times and therefore performance for their application on a given hardware platform.

## 4.1 Characteristics of Hardware and Application

To calculate the response time for a request some key characteristics of both hardware and software have to be known. These values can be considered as input parameters for the performance model. These parameters can be split up into parameters that characterize the computing platform and parameters characterizing the application program. For the hardware and the operating system used to run the application the following values have to be known or measured:

$n_{\text{hwt}}$ The number of HWTs in the system

$b_c$ The total bandwidth available for communication between two HWTs; This number already includes information about the communication architecture, e.g. about redundant wiring. Unit: B/s

$t_{\text{prop}}$ The average propagation delay of the communication infrastructure; Unit: s

$t_p$ The time until a request for a single HWT is processed by the HWTM, assuming at least one HWT is available; Unit: s

$t_{\text{return}}$ The time it takes for the HWTM to process the return of a single HWT; Unit: s

The parameters that depend on the application running on top of the operating system are:

$r_r$ The rate at which requests for new HWTs are issued by the application; Unit: 1/s

$t_{\text{calc}}$ The duration of the calculations performed by a thread; Afterwards its HWT is returned. Unit: s

$\bar{n}$ The average amount of requested HWTs per request

$r_c$ The rate at which a single thread communicates with other threads, while it is working; This number does not include requests to the HWTM. Unit: 1/s

$\bar{s}_c$ The average size of communication messages; Unit: B

The values of the request rate and the calculation time of a thread are assumed to be poisson distributed. This is a common assumption for requests arriving into a queue [All90] [Med02] and is also used to model several other request patterns, e.g. requests to web servers [CZ03]. Their expectancy values are $r_r$ and $t_{\text{calc}}$.

The hardware thread management architecture can be configured by the following parameters:

$n_q$ The size of the request queue at the HWTM used as a waiting queue for requests

$n_{\text{managers}}$ The amount of HWTM instances used; This number describes the degree of distribution.

## 4.2 Basic Design

A HWTM is a piece of software that runs on a single HWT. This HWT is completely dedicated to the management of other HWTs and does not execute any application code, so there are no context switches that would impact the performance of the management code. A similar concept was presented in [HRY+08], where cores are dedicated for only running kernel code and executing system calls on behalf of other cores to reduce the need of context switches and therefore reduce the probability of cache misses and especially translation lookaside buffer (TLB) invalidations. Each HWTM is assigned a set of HWTs it is responsible for.

When an application requests additional threads, this request is always sent to the HWTM responsible for the HWT the requesting code is running on. A reference to this HWTM can be stored in thread local storage (TLS) for fast access. The HWTM's task then is to accumulate the number of HWTs specified in the request and return their handles to the requesting thread. Since only available HWTs can be used for calculations, it has to keep track of the current state of its managed HWTs and only pass out available ones, that are in turn marked as occupied. Upon the return of a HWT it is marked as available again. To provide the application with these operations the HWTM exposes two functions: `request_workers(n)` to request $n$ HWTs and `return_worker(handle)` to return the HWT with the given handle.

Since the HWT running the management code does not execute any application code, for a fixed number $n_{\text{managers}}$ of HWTMs the number of worker HWTs available to the application is given by

$$n_{\text{workers}} = n_{\text{hwt}} - n_{\text{managers}} \tag{4.1}$$

The HWTM developed in this thesis supports distributed processing of requests and therefore is capable of running multiple HWTM instances on different HWTs in parallel. However, these instances have to interact with each other in order to synchronize their state information. In the following, first the operation of a system with only one HWTM instance is described to model the basic mode of operation. Therefore,

the number of workers is

$$n_{\text{workers}} = n_{\text{hwt}} - 1 \tag{4.2}$$

Afterwards, this model is extended to multiple HWTM instances, that share the workload.

## 4.3 Considering a Single Hardware Thread Manager

When the HWTM receives a new request for one HWT it takes $t_p$ to process the request, if at least one HWT is available. There are several tasks it has to complete to process the request:

1. Search for an available HWT in the list of managed HWTs. The first available HWT in the list is selected and marked as occupied.

2. Assign a unique identifier to each hardware thread. In case of only one hardware thread manager, this is a trivial task and has a constant runtime, e.g. using the value of an increasing counter. Since the HWT ID can be used as an unique identifier, this is trivial for one-to-one mappings of HWTs to threads, as well.

3. Optionally create a (software) thread on the selected HWT.

4. Accumulate this information into a structure that is returned.

5. The next available HWT is searched and a pointer to it is stored for later use.

The time for processing a request for $n$ HWTs is determined by

$$t_p(n) = n \cdot t_p. \tag{4.3}$$

When a thread finishes its calculations, it sends a message to the HWTM. The manager puts the HWT back to the free list, which takes a constant runtime of $t_{\text{return}}$. The HWT then can again be allocated to an other request. When the HWTM is done processing a request, it continues with processing the next one in the waiting queue.

### 4.3.1 Number of Available Hardware Threads

The number of HWTs that are currently available throughout the system can be modeled as a continuous time markov chain (CTMC) with the state numbers representing the current number of available HWTs. The rate at which threads are requested is independent of the number of currently working threads. Therefore, a transition from state $i$ to $i-1$ takes place at a rate of $r_r \cdot \bar{n}$. The rate of HWTs returned to the manager depends on the number of currently working threads, since in a system with more active threads more threads finish their work per time interval. The rate of thread returns can be calculated as $(n_{\text{workers}} - i)/t_{\text{calc}}$ for $i$ available threads. This is the rate of the transition from state $i$ to $i+1$ in the CTMC. Let $q_{i,j}$ denote the rate of a state transition from state $i$ to state $j$, then the number of available hardware threads is described using

$$q_{i,i-1} = r_r \cdot \bar{n} \qquad \text{for all} \quad 1 \leq i \leq n_{\text{workers}} \tag{4.4}$$

$$q_{i,i+1} = \frac{n_{\text{workers}} - i}{t_{\text{calc}}} \qquad \text{for all} \quad 0 \leq i \leq n_{\text{workers}} - 1. \tag{4.5}$$

These transition rates are encapsulated in the transition rate matrix Q. To obtain a valid CTMC, the values on the major diagonal ($q_{i,i}$) have to be chosen such that the rows of $Q$ add up to zero. All values of $Q$ not located on the major or a minor diagonal are set to zero. The analysis of the steady state distribution of this markov chain provides a probability distribution of the number of available HWTs in the whole system. The probability of $x$ HWTs being available (or of $n_{\text{workers}} - x$ hardware threads working) is denoted by $\pi_x$, which can be computed as the steady state distribution of the CTMC. This solution is obtained by solving the equation

$$\pi_x Q = 0 \tag{4.6}$$

with respect to $\pi_x$ according to [Nor98]. The solution has to meet the condition, that all entries in $\pi_x$ add up to one.
Assuming $n_{\text{workers}} = 3$, $r_r \cdot \bar{n} = 1$ and $t_{\text{calc}} = 2$, the matrix Q yields

$$Q = \begin{pmatrix} -1.5 & 1.5 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -1.5 & 0.5 \\ 0 & 0 & 1 & -1 \end{pmatrix} \tag{4.7}$$

which corresponds to the illustration in figure 4.1. The steady state for this system then is

$$\pi_x = \begin{pmatrix} 0.21 & 0.32 & 0.31 & 0.16 \end{pmatrix}. \tag{4.8}$$

So, for this example, the probability of all threads being busy is 0.21, while all threads are available with a probability of 0.16.

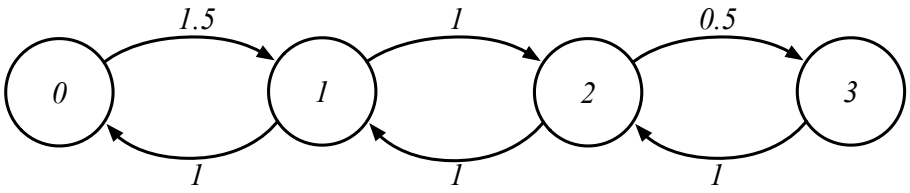



**Figure 4.1:** Exemplary CTMC for modeling a system with $n_{\text{workers}} = 3$, $r_r \cdot \bar{n} = 1$ and $t_{\text{calc}} = 2$; States are labeled with the number of available HWTs in the system, transitions are labeled with transition rates.

### 4.3.2 Communication Latency

For calculating the total time from issuing the command to send a message until it is received, three major parts have to be considered:

1. **Transmission delay:** The time it takes for a message to be serialized onto the transmission medium. It is calculated as

$$t_{\text{trans}} = \frac{\text{message size}}{\text{link bandwidth}}.$$

2. **Propagation delay:** The time a message is being transmitted through the link. It is determined by

$$t_{\text{prop}} = \frac{\text{link length}}{\text{propagation speed}}.$$

3. **Queueing delay:** The time a message has to wait for transmission. It depends on the link load and is given by

$$t_{\text{wait}} = \frac{1}{\frac{\text{link bandwidth}}{\text{message size}} - \text{message rate}}.$$

For the calculations of the communication latency a ring bus connecting all hardware threads is assumed. In order to consider redundant wiring, the value used as total available bandwidth could be adapted, e.g. for a double ring bus, $b_c$ can be set to double the bandwidth of a single link. The latency of communication between a HWT and the HWTM heavily depends on the amount of communication that is triggered by the application for its own purposes.

For requesting HWTs, messages are sent to the HWTM at a rate of $r_r$. Analogously the message rate for returning threads is given by utilizing the probability of a certain number of threads being active:

$$r_{\text{returns}} = \sum_{i=0}^{n_{\text{workers}}} \pi_i \cdot \frac{n_{\text{workers}} - i}{t_{\text{calc}}} \tag{4.9}$$

The rate at which the running application sends messages for communication between worker hardware threads is determined by

$$r_{\text{other}} = \sum_{i=0}^{n_{\text{workers}}} \pi_i \cdot (n_{\text{workers}} - i) \cdot r_c. \tag{4.10}$$

Therefore, the total rate of messages being transmitted over the interconnect is given by

$$r_{\text{total}} = r_r + r_{\text{returns}} + r_{\text{other}}. \tag{4.11}$$

The propagation delay, i.e. the time of a packet being "in flight" on the link, is assumed to be a constant in the system and is therefore given as a fixed parameter.

Using these variables, the mean time needed for transmitting an additional message is calculated as

$$t_c = t_{\text{trans}} + t_{\text{prop}} + t_{\text{wait}} \tag{4.12}$$

$$t_c = \frac{\bar{s}_c}{b_c} + t_{\text{prop}} + \frac{1}{\frac{b_c}{s_c} - r_{\text{total}}}. \tag{4.13}$$

Equation (4.13) only holds, if $r_{\text{total}} \leq \frac{b_c}{s_c}$, or in other words, if the link is capable of transporting all the messages that need to be sent. If this condition is violated, the mean queueing time approaches infinity and no reasonable transmission is possible.

### 4.3.3 Waiting Queues and Total Response Time

To derive the total response time it takes for the application from issuing a thread creation request experiences until a handle is returned, the state of the request queue of the hardware thread manager has to be known. For describing queues Kendall's notation is used. This notation also includes the possibility to describe random arrival times with a certain mean value. The queueing process is described in Kendall's notation as a $\mathrm{M}(r_r)/\mathrm{M}(\frac{1}{\bar{t}_p})/1/n_q$ queue. This denotes a queue with random arrival (poisson process) at an average rate of $r_r$, a random service time with a mean of $\frac{1}{\bar{t}_p}$, one server (the hardware thread manager itself) and a queue size of $n_q$. Random arrival times are chosen, because the arrival of requests depends on the implementation of the application. Poisson processes are the most common choice when describing such queues [Med02]. Random service time is a reasonable choice, since it depends on the amount of hardware threads requested, which is chosen by the application. Although the individual service times are determined by a random process, the average service time for an arbitrary request of $\bar{t}_p = t_p(\bar{n})$ is known.

According to [All90], the average waiting time of a request in a M/M/1/N queue with average arrival rate of $r_r$, average service time $\bar{t}_p$ and capacity $n_q$, is given by

$$t_{\text{queue}} = \frac{L}{r_r(1 - p_{n_q})} \tag{4.14}$$

with $L$ denoting the expected steady state number of occupied waiting slots. $L$ is calculated as

$$L = \begin{cases} \frac{a}{1-a} - \frac{(n_q+1)a^{n_q+1}}{1-a^{n_q+1}} & \text{if } a \neq 1 \\ \frac{n_q}{2} & \text{if } a = 1. \end{cases} \tag{4.15}$$

Here $a$ denotes the occupancy rate of the HWTM, i.e. the fraction of time spent processig requests. It is given by

$$a = \bar{t}_p \cdot r_r. \tag{4.16}$$

The case $a = 1$ occurs, if requests arrive exactly as fast as they are processed. The probability $p_{n_q}$ of the queue being filled completely is determined by

$$p_{n_q} = \begin{cases} \frac{(1-a)a^{n_q}}{1-a^{n_q+1}} & \text{if } a \neq 1 \\ \frac{1}{n_q+1} & \text{if } a = 1. \end{cases} \tag{4.17}$$

A request for $n$ threads being sent to the hardware thread manager experiences a total response time $t_{\text{request}}$ from issuing the request until a handle is received:

$$t_{\text{request}}(n) = t_c + t_{\text{queue}} + t_p(n) + t_c \tag{4.18}$$

This includes the time for two messages being sent, used for request and response, the time the request spends in the waiting queue and the time for the actual processing of the request by the HWTM.

### 4.3.4 Queue Overflow

Assuming that there are only as many HWTs requested as there are HWTs in the system, the queue of the HWTM needs to be as large as the number of HWT for avoiding an overflow of the queue:

$$n_q \geq n_{\text{workers}} \tag{4.19}$$

This would cover the case where each HWT is requested simultaneously in a single request each. Depending on the total number of HWTs, this approach could lead to a very large queue that could be nearly empty most of the time. The size of the queue could make the amount of memory used by the hardware thread manager too large to permanently stay in the local cache for a large number of hardware threads, which leads to increasing runtimes due to cache misses and time spent for accessing memory. Therefore, a smaller queue size might be a better choice. In this case, a mechanism notifying the requesting thread, whether a request was dropped, is required. The application then may transmit the request again at a later time.

To determine the probability of a request being dropped, the steady state of the queue must be analyzed. In [BG00] a way to calculate the probability of a queue being filled with $0 \leq x \leq n_q$ elements is presented. In a queue with capacity $n_q + 1$, the probability of this auxiliary queue being filled completely, is the same as a queue with capacity $n$ having a queue overflow. The probability of the auxiliary queue with capacity $n_q + 1$ being filled completely can be calculated using equation (4.17):

$$p_{\text{overflow}} = \frac{(1-a)a^{n_q+1}}{1-a^{n_q+2}} \tag{4.20}$$

It would also be possible to derive the time variant behavior of the queue and thereby calculate the time variant probability of a queue overflow, as presented in [GBG02]. However, as the considered system is long-running, the transient phase, which is

characterized by the time variant probability, until the steady state is reached can be neglected.

A discarded request that has to be transmitted again, leads to an increased usage of communication bandwidth and an increased duration of thread creation.

### Effects Caused by Using Small Queues

**Loss** In case of a queue overflow causing a request being discarded, this event has to be reported back to the requesting node. The request has to be issued again after some waiting time $t_{\text{retransmit}}$. As a result, the total delay of the request increases by $2t_c + t_{\text{retransmit}}$. For this reason equation (4.18) has to be extended to model this behavior. Using $p_{\text{overflow}}$ to denote the probability that a request has beed discarded due to a full queue, as previously described, the total request time is given by

$$t_{\text{request}}(n) = t_c + t_{\text{queue}} + t_p \cdot n + t_c + (2t_c + t_{\text{retransmit}}) \cdot \sum_{k=1}^{\infty} p_{\text{overflow}}^k. \tag{4.21}$$

The summation index $k$ describes the number of times a request is retransmitted. Since $|p_{\text{overflow}}| < 1$ this can be simplified by applying the geometric series:

$$t_{\text{request}}(n) = t_c + t_{\text{queue}} + t_p \cdot n + t_c + (2t_c + t_{\text{retransmit}}) \cdot \left( \frac{1}{1 - p_{\text{overflow}}} - 1 \right) \tag{4.22}$$

This equation also includes the cases, where the request is discarded multiple times in a row.

**Benefit** The main benefit of using a queue with size $n_q \leq n_{\text{workers}}$ is the reduced amount of memory required to store it. If the queue is too large to fit into the cache of the hardware thread running the hardware thread manager, a lot of cache misses may occur when adding elements to the queue at the end and when fetching or removing them from the front. The maximum queue length, where the complete code of the HWTM and its entire waiting queue fit into the cache of a single HWT, depends on the implementation of the management architecture and the hardware used for running it and has to be determined separately.

## 4.4 Introducing a Second Management Instance

In this section, a system with two management instances is considered. Since work is split up between these instances, the workload for each instance is reduced. The two instances share the workload, because requests are directed to the HWTM responsible for the requesting HWT and requests are assumed to be issued equally distributed over the available HWTs.
In the following, the case of only two management instances is considered. In this case, one instance immediately knows, which instance is responsible for all available hardware threads not managed by itself, namely the other instance. However, where possible, equations are given in a way that also applies for an arbitrary number $n_{\text{managers}}$ of hardware thread managers in the system. Initially, the state synchronization required for more than two management instances is neglected and introduced afterwards.

When introducing an additional hardware thread management instance, dedicating a second HWT completely for that task, the number of hardware threads available to the application is reduced by one, since $n_{\text{managers}}$ is increased by one. These worker hardware threads are then split up into $n_{\text{managers}}$ groups. Each group is assigned to one hardware thread manager. Therefore, each of the management instances is, as an upper bound, in charge of

$$n_{\text{managed}} = \left\lceil \frac{n_{\text{workers}}}{n_{\text{managers}}} \right\rceil \tag{4.23}$$

HWTs. The request rate, at which new requests arrive at a HWTM, is also reduced, because of the reduction of managed hardware threads. The rate $r_{r,1}$, denoting the rate at which requests arrive at a single management instance, is given by

$$r_{r,1} = \frac{r_r}{n_{\text{managers}}}. \tag{4.24}$$

In order to minimize communication distance to the HWTM, each HWT is associated to the closest management instance and directs all requests to this instance. Further, all responses are received from this instance. The management instances should be distributed uniformly over the communication infrastructure to ensure minimal communication distances between threads and HWTMs and a fair division of HWTs between the management instances.

Sometimes the HWTM instances may have to communicate with each other. Although messages exchanged between worker HWTs and a management instance usu-

ally are the more common message type, some requests may have to be delegated. The basic communication between two HWTM instances is covered in the following section.

### 4.4.1 Communication

If a request arrives at a HWTM and it is not able to satisfy it immediately, because it does not have enough HWTs available, the request has to be delegated. For this delegation, the original HWTM issues a request for the remaining number of HWTs and sends it to the other manager. After it receives a response, the local and remotely computed results are combined and sent back to the original requester.

The amount of random communication between different worker threads, which occurs at a rate of $r_c$, does not change when introducing an additional management instance. However, additional messages may need to be sent in order to request hardware threads from the other manager, if necessary. The number of additional messages required for this task, is derived in the following section.

### 4.4.2 Inability to Satisfy a Request

A request of a thread to get $n$ HWTs cannot be fulfilled, if the number of available HWTs at the management instance is smaller than $n$. To derive the probability of a single management instance having less than $n$ hardware threads available, an assumption about the distribution of requests is needed. Since the considered system is long running, it is assumed, that, after some initial phase, the requests for new threads arrive randomly at each of the management instances. In general the probability for each instance is $1/n_{\text{managers}}$, in this case of two management instances this equals 50%.
The inability of a HWTM to satisfy a request is described by a hypergeometric distribution and the probability of a management instance being unable to fulfill a request, while in total $x$ HWTs are available, is given by

$$p(n,x) = \sum_{k=0}^{n-1} \frac{\binom{n_{\text{managed}}}{k}\binom{n_{\text{workers}}-n_{\text{managed}}}{x-k}}{\binom{n_{\text{workers}}}{k}}. \tag{4.25}$$

This can be pictured as a random draw, where for each available HWT a manager is randomly chosen to be responsible of it. Therefore, a hypergeometric distribution with population size $n_{\text{workers}}$, number of success states $n_{\text{managed}}$, number of draws $x$

and number of observed successes $k$ is used here. Combining this equation with the previously derived probabilities $\pi_x$ of $x$ HWTs being available, yields

$$p(n) = \sum_{x=0}^{n_{\text{workers}}} \pi_x p(n, x) \tag{4.26}$$

to obtain the overall probability of a management instance being unable to satisfy a request for $n$ HWTs and having to split up the request.

### 4.4.3 Total Response Time

If a management instance is unable to fulfill a request, the request is redirected to the other management instance. To acquire the remaining HWTs, the initial management instance has to request those from the other instance, wait for a response, and deliver the results back to the requesting thread. So, it has to wait as long as a normal thread, when requesting HWTs. Additionally, the response of the other instance has to pass through the message queue. In combination with (4.22) the total request time is given by

$$t_{\text{request},2} = t_{\text{request}}(n) + p(n)(2 \cdot t_{\text{queue}} + 2 \cdot t_c). \tag{4.27}$$

Since the time for actually processing the request $t_{\text{request}}(n)$ does not depend on the size of the delegated part of the request, it can be included all at once. This means, that a thread requesting hardware threads has to wait longer until the threads are received. However, the management instance can handle other requests in the meantime.

### 4.4.4 Queueing Time

The requests for threads are split up between the instances, reducing the number of requests for a single instance. However, additional requests are now generated, if one instance requests threads from another one. This case occurs with probability $p(n)$, so the additional request rate is $p(n)r_r$. The time a request spends in the queue is calculated, by replacing $r_r$ with $p(n)r_r + r_{r,1}$ in equations (4.14) - (4.17).

## 4.5 Increasing the Number of Management Instances

A new problem arises, if the number of management instances is increased beyond two. If only two management instances are present and one instance cannot fulfill a request using only the HWTs managed by itself, it sends a message to the other management instance. In case of more than two management instances, it is not obvious, which instance has which amount of available hardware threads and which instance should be contacted. Therefore, replication of the current state of the management instances is required. When the number of available HWTs at a management instance changes, the new number of available HWTs is propagated to all other HWTMs by sending out a broadcast message. Then every management instance can determine the HWTM with the most available HWTs and request them from this instance. This HWTM is selected, because it yields the highest probability to fulfill the request, after it passes the queue. In this way, each HWTM has a local list of the state of its managed HWTs and a list of the amount of available HWTs at the other instances. So each HWTM has a fine-grained view of its own HWTs and only accumulated numbers for the other instances.

### 4.5.1 Mode of Operation

When processing a request for HWTs, a management instance now carries out the following protocol

1. Assign locally available hardware threads to the request
2. If there are not enough hardware threads available
    a) Find the hardware thread manager with the largest amount of available hardware threads
    b) Request additional hardware threads from this management instance
    c) Subtract number of requested hardware threads from stored numbers
    d) If there are still hardware threads missing, repeat from 2a)

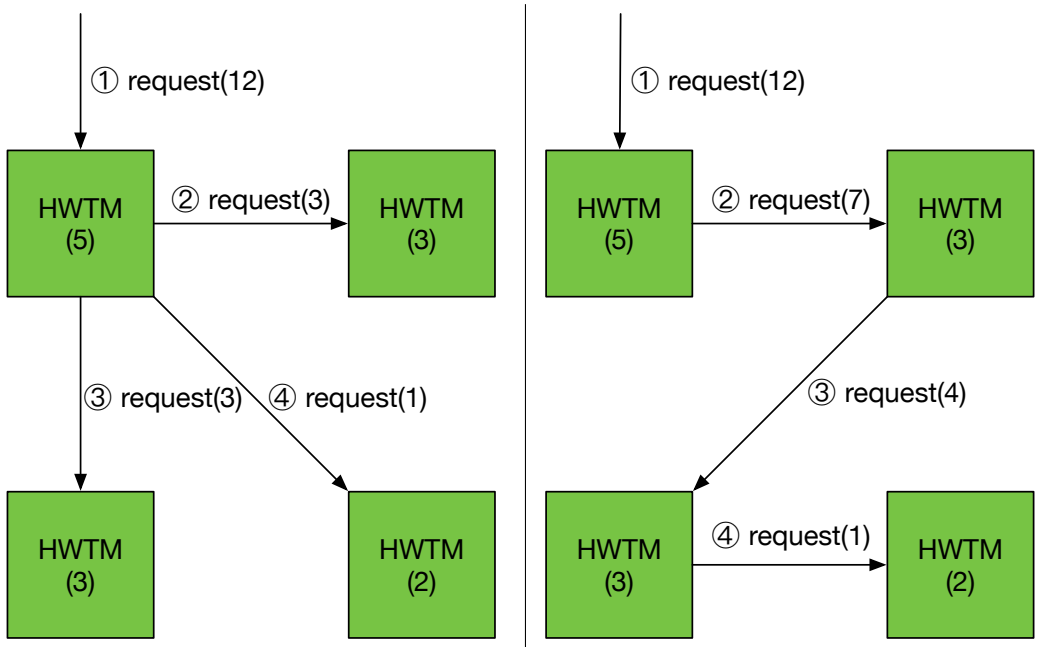
3. Update the number of locally available HWTs at the other managers by sending them an update message

If a HWTM receives a delegated request from another management instance, it may not being able to fulfill the request. This is the case, if HWTs have been assigned to requests between the last update of the local state and the processing of the delegated request. The amounts of available HWTs stored at other HWTMs only represent the current view of state, not the state present at the time a new request is being processed. In the case of a HWTM not being able to fulfill a delegated request, it assigns as many HWTs to it as possible and returns them to the requesting manager. The number of HWTs returned may be zero. The requesting instance keeps track of sent out requests and issues additional requests, if a previous request was not fulfilled completely. To ensure using the most recent numbers for delegation decisions, state updates from other management instances have to be processed at high priority.
The algorithm used for request delegation is an iterative one, where the first HWTM is responsible of contacting the other managers and collecting the results. The same functionality could also be implemented as a recursive algorithm, where the first HWTM delegates the remaining request to another manager instance completely. This instance then in turn assigns as many hardware threads as possible and then delegates the remainder again. However, this recursive algorithm has drawbacks, as it needs to deal with cycle resolution and more complex communication schemes. Therefore, the iterative implementation is considered to be more performant and simpler while delivering the same functionality.

The two delegation schemes are illustrated in figure 4.2. In both cases a request for 12 HWTs cannot be satisfied by the HWTM contacted by the application and has to be delegated. Figure 4.2a shows the case of iterative delegation, where the first HWTM contacts each of the other managers and requests a small amount of HWTs from them. Figure 4.2b illustrates recursive delegation, where the remaining request is completely delegated to one other instance. In this case, each instance has to carefully avoid forming a cyclic delegation pattern, because that would break the delegation algorithm.

### 4.5.2 Number of Delegation Messages

As described previously in this chapter, a manager sends messages to all other managers after it is done processing a request. So, assuming the system is capable of processing all the requests, messages are sent out at a rate of $r_r$. Since this messages need to be distributed to all other managers, the worst case number of messages is $r_r(n_{\text{managers}} - 1)$.




(a) Iterative delegation scheme (b) Recursive delegation scheme

**Figure 4.2:** Comparison of different delegation schemes; A request for 12 HWTs cannot be satisfied by the first HWTM and is delegated to other management instances.

Additionally, delegation messages are sent whenever a request cannot be fulfilled by a manager. This situation occurs at a rate of $p(\bar{n})r_r$. For this case, it has to be determined how many delegation requests have to be sent out. On average, there are $(1-p(x)) \cdot (n_{\text{managers}} - 1)$ of the other management instances with exactly $x$ hardware threads available.

To determine the number of delegation messages, it is helpful to look at the number of available hardware threads at a management instance as a random variable. This random variable $P_i$ represents the number of available HWTs at management instance $i$ and has the probability mass function $f_P(x) = 1 - p(x)$. The probability of a set of $n$ management instances being able to fulfill a request for $h$ HWTs then is

$$A(n,h) = \sum_{P_1,\ldots,P_n,\sum_i P_i \geq h} f_P(x = P_1) \cdot \ldots \cdot f_P(x = P_n) \tag{4.28}$$

$$= \sum_{P_1,\ldots,P_n,\sum_i P_i \geq h} \prod_{k=1}^{n} f_P(x = P_k). \tag{4.29}$$

This describes the probability of more than $h$ hardware threads being available, when $n$ management instances are being considered. To obtain the summation indices, all possible amounts of available hardware threads, where in total more than $h$ hardware threads are available, have to be considered. The function $A(n, h)$ is monotonically increasing for variable $n$ and describes this probability with $n$ or less management instances being considered. To obtain the probability to require exactly $n$ management instances, using a fixed request size of $h = \bar{n}$, the cumulative function (4.29) is transformed into a non-cumulative one by calculating the discrete derivative

$$A(n) = A(n, \bar{n}) - A(n-1, \bar{n}). \tag{4.30}$$

This results in the expected number of required delegations given by

$$n_{\text{delegations}} = \left( \sum_{k=1}^{n_{\text{managers}}} A(k) \cdot k \right) - 1. \tag{4.31}$$

This is equal to the expectancy value of A with respect to n, which is the average number of management instances needed to fulfill a request. Since the request does not need to be delegated to the instance originally processing it, 1 has to be subtracted to gain the number of delegations.

### 4.5.3 Total Response Time

For each request an average of $n_{\text{delegations}}$ delegation messages are sent out, which are each answered with another message. Therefore, the rate of messages caused by delegation is $2 \cdot r_r \cdot n_{\text{delegations}}$.

In total, the additional rate of messages, that need to be sent to support the introduced protocol, including delegations and updates, are given by

$$r_r(n_{\text{managers}} - 1) + 2 \cdot r_r \cdot n_{\text{delegations}}. \tag{4.32}$$

This leads to an increased amount of communication for supporting the delegation of requests. However, by increasing the number of management instances, the average communication distance between a hardware thread and its corresponding management instance is reduced.

The total time a request takes to be processed by $N$ HWTMs is

$$t_{\text{request},N} = t_{\text{request}}(n) + p(n) \cdot t_{\text{queue}} + (r_r(n_{\text{managers}} - 1) + 2 \cdot r_r \cdot n_{\text{delegations}}) \cdot t_c. \quad (4.33)$$

This response time consists of the time for the actual processing, as explained prviously, the time for queueing at remote HWTM instances, if applicable, and the time for communication due to delegation and update messages.

### 4.5.4 Queueing Time

Since delegated requests are placed into the queues of different management instances, the queues' lengths increases. With a probability of $n_{\text{delegations}}/n_{\text{managers}}$ a delegated request is placed into a specific queue. Therefore, the request rate for a single queue is $r_r/n_{\text{managers}} + r_r \cdot n_{\text{delegations}}/n_{\text{managers}}$. Using this request rate, the queue length is calculated as described in section 4.3.3 using equations (4.14) - (4.17).

## 4.6 Determining the Best Configuration

Now the information for selecting the best configuration of management instances is available. The gain in response time, when including an additional manager is given by the relative increase in response time using equation (4.33)

$$\text{gain}(n) = \frac{t_{\text{request,n}-1} - t_{\text{request,n}}}{t_{\text{request,n}-1}}. \quad (4.34)$$

The loss in computing power is determined by

$$\text{loss}(n) = \frac{1}{n_{\text{workers}}}. \quad (4.35)$$

As long as the gain is higher than the loss, it is reasonable to include additional management instances. Therefore, the number $n_{opt}$ with

$$n_{opt} = \max_n \{n \in \mathbb{N} : \text{gain}(m) > \text{loss}(m)\, \forall m \leq n, m \in \mathbb{N}\} \quad (4.36)$$

is selected as the optimal number of management instances.

This number of management instances is considered to provide the best tradeoff between response time and loss of computational power by not using individual HWTs

for computations by the application. Exemplary results were verified using a custom implementation of the proposed HWTM architecture that is presented in chapter 6 and the platform used for evaluation purposes is described in chapter 5.

# 5 Evaluation Platform

In order to validate and evaluate the developed performance model, a distributed hardware thread manager was implemented and integrated into the Many Threads Operating System (MyThOS), which is being developed at the Institute of Information Resource Management at the University of Ulm. The basic principles of MyThOS are explained in section 5.2. This operating system can be run on several platforms. For this thesis an Intel® Xeon Phi™ was chosen as the evaluation platform, since it offers a high degree of parallelism on a single CPU. Its architecture is described in section 5.1.

## 5.1 Hardware: Xeon Phi

The evaluation of the developed performance model was carried out on an Intel® Xeon Phi™ 5110P, which features 60 CPU cores with four Hardware Threads per core, adding up to a total of 240 hardware threads. It consists of a single PCIe-card, which is plugged into a standard Xeon server and encloses a CPU, its main memory and an interconnect for communication across the chip. Although it can be used as a coprocessor for offloading jobs from the server onto the Xeon Phi, it also can be used as an independent computer, running its own operating system and software. However, the image of the operating system is still provided by the host server, which also is able to boot or reset the coprocessor card. It also is capable of reading the contents of the Xeon Phi's main memory. The CPU cores of the Xeon Phi are able to communicate with each other using an on-chip interconnect. The memory of the Xeon Phi is also accessed using this interconnect. These individual components are further described in the following. A detailed documentation of the Xeon Phi's architecture is available from Intel [Int13].

### 5.1.1 CPU Core Architecture

The Xeon Phi provides a large amount of CPU cores on a single chip, which makes it suitable for massively parallel workloads, without the need of communication across CPU boundaries. However, there are some drawbacks of the CPU cores on a Xeon Phi compared to state of the art server CPUs. First, the Xeon Phi's CPU is limited in its clock frequency of 1.053 GHz and does not offer common instruction sets, e.g. MMX, AVX and SSE. Second, it also does not offer out-of-order pipelines, that would allow the CPU to dynamically rearrange instructions to avoid blocking because of instructions whose parameters have not been computed yet.

The L1-cache of each core consists of a 32 kB instruction cache and a 32 kB data cache. The L2-cache of each core has a size of 512 kB and can be accessed by other cores on demand for fast access to data available in the local L2-cache. By using this architecture, each core has its own L2-cache, but has no need to access the main memory for data that is already present in the L2-cache of another core. A ditributed tag directory is used for cache coherency.

Each CPU core consists of four hardware threads that share a 64-bit floating point unit and a 512-bit vector unit. The core utilizes time multiplexing to switch between these hardware threads. However, the scheduler is not able to issue instructions from the same hardware thread in two consecutive clock cycles. Due to this limitation a core running only one hardware thread can only utilize the core by 50% in the best case. Two hardware threads could fully utilize the core, if no cache misses or other blocking operations occurred. Since this is a unrealistic scenario, four hardware threads per core are used.

### 5.1.2 Interconnect

The Xeon Phi's CPU cores are connected by a ring interconnect, which provides two rings connecting all cores, one traveling each direction. This interconnect is illustrated in figure 5.1. The individual cores are connected to the ring interconnect by a core ring interface that also hosts the L2-cache for each core. This interface allows its core to send and receive messages over the interconnect. Each of the two directions of the ring consists of three independent rings: A high-bandwidth data ring to transfer large amounts of data between CPU cores, an address ring to transfer memory addresses and an acknowledgement ring to transfer flow control messages. The address ring is furthermore used for replication of remote L2-caches as described in the previous section. A distributed tag directory is used for keeping the L2-cache fully coherent.

Although the Xeon Phi offers a separate ring for each direction, the system does not necessarily choose the shortest path for a message. When a message is sent over the interconnect, there are no guarantees for which ring is chosen for the message and the system may choose an arbitrary ring for this communication. In the worst case a message between two neighboring cores passes the whole ring and occupy it completely during that time.

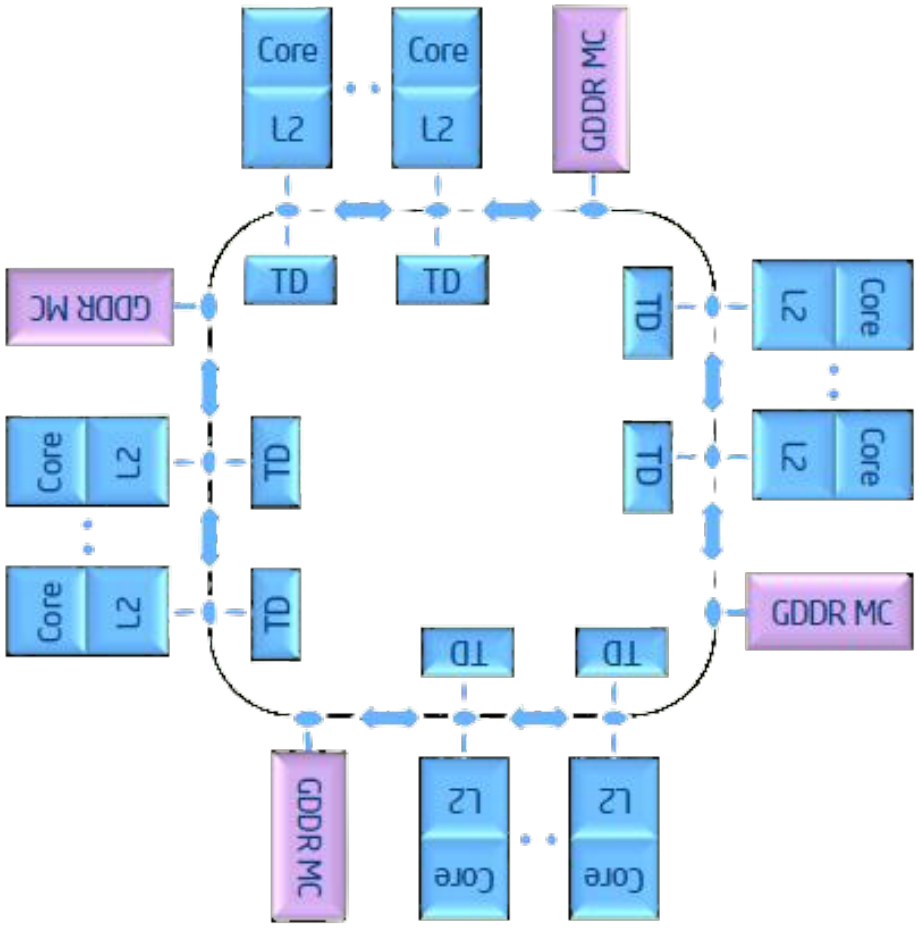



**Figure 5.1:** Architecture of the Xeon Phi's ring interconnect; Source: [Int12]

The ring interconnect does not only connect the cores with each other, it also is used for memory access, which is provided by memory controllers that are symmetrically distributed around the ring. Memory addresses are evenly distributed among the memory controllers, which prevents hotspots and therefore ensures uniform access patterns.

## 5.2 Operating System: MyThOS

The Institute of Information Resource Management is developing an operating system called *Many Threads Operating System (MyThOS)*, that was used as a foundation for implementations and measurements in this thesis. MyThOS is an operating system based on a microkernel. Its main goal is to equip instances of the operating system with only the resources it requires, providing as much memory and locality to each thread as possible. An example is a worker thread in a common multithreaded application, which does not need to be capable of managing instances of the operating system. It also does not need to be able to process every system call locally if it can be delegated to another instance. In this way, the environment encapsulating a thread is kept minimal, providing the thread with unpolluted caches, which reduces the number of cache misses and therefore provides maximum performance.

MyThOS is designed to quickly allocate and instantiate large amounts of threads and manage large distributed computing infrastructures with many CPU cores and hardware threads. Therefore, it is an ideal candidate for the integration of a hardware thread manager, like the one described in section 4, to keep track of available and occupied computing resources in the system. Another benefit of choosing MyThOS as an evaluation platform is its support for the Xeon Phi coprocessor, that provides an amount of HWTs requiring managedistributed management of computing resources.

The execution environment for applications running on MyThOS is managed by the class `ExecutionContext`. Its main tasks are the allocation of stacks for kernelspace and userspace, management of execution state and the deployment of tasks to individual hardware threads. For applications MyThOS offers system call interfaces for instantiation of new threads and deployment of those threads onto hardware threads:

```
1 void* newExecutionContext(size_t location, void* stacktop, size_t
     maxstacksize);
2 size_t callECasync(void* ec, void (*funptr)(void*), void* dataptr)
```

The first system call instructs MyThOS to create a new ExecutionContext on the hardware thread with LAPIC-ID `location` and allocate and assign a userspace stack for it at location `stacktop` with size `maxstacksize`. The second system call makes the previously created ExecutionContext run the function `funptr` and passes `dataptr` as a argument to that function.

MyThOS offers its own mechanisms for communication among HWTs. During the boot process of the operating system two queues are created for each HWT, a *default queue* and an *idle queue*, which are used to assign tasks to the corresponding HWT. Tasks placed in the default queue are processed one after the other according to the

first in, first out (FIFO) strategy. If no tasks are present in the default queue, tasks placed in the idle queue are processed in the same way. The queues are located in the main memory of the system and can be accessed by all HWTs in the system. Due to this organization, all communication inside the kernel passes through main memory. The queues are accessed by calling one of the functions `getDefaultQueue()` or `getIdleQueue()` to get the queues of the own HWT or by calling one of the functions `getRemoteDefaultQueue(id)` or `getRemoteIdleQueue(id)` to get the queues of the HWT with the given ID.

Tasks placed in these queues are realized by using tasklets. Tasklets are essentially small pieces of code that are executed without any preemption. This has two main advantages: First, the code inside a tasklet can rely on being executed all at once and does not have to deal with effects related to preemption. Therefore, it may require less mutexes to prevent race conditions or deadlocks. Second, the scheduler in MyThOS can be kept very basic, since no bookkeeping of time slices or functionality for preemption has to be implemented. It therefore is able to provide very high performance. However, a developer has to be careful not to include any blocking operations inside a tasklet, since they would prevent the HWT from running any code while blocking, since the tasklet would not be preempted and no other tasklet will be run while the first one blocks. As a single tasklet can block its hosting HWT forever, this design relies on trust, assuming no harmful code is executed inside the kernel.

Tasklets are placed inside a queue by calling the queue's `push` method, which takes either a function pointer or a functor as a parameter. This function pointer or functor is wrapped into a tasklet, which is placed into the queue. The scheduler takes a tasklet out of the queue, executes the corresponding task and continues with the next tasklet after the execution is finished.

The Xeon Phi and MyThOS form the platform for the implementation of the presented HWTM. This implementation is presented in the next chapter.

# 6 Implementation

To verify the performance model presented in chapter 4 an implementation of the proposed management architecture was developed. The two parts, the performance model to provide reference values and the HWTM to verify these values, were developed separately. First, the implementation of the performance model is presented in section 6.1. Afterwards, the implementation of the HWTM based on MyThOS and the Xeon Phi is described in section 6.2.

## 6.1 Performance Model

In order to determine the optimal number of instances of the distributed hardware thread manager (HWTM) described in chapter 4 the presented performance model was implemented in MATLAB [MAT15]. The aim of this implementation is to determine the response time for requests to the HWTM architecture for different numbers of HWTM instances and given parameters. These parameters, which were described in section 4.1, are used as an input to the program. While some of the formula of the performance model can be implemented in a straight forward way, others have to be rearranged or transformed to reduce their memory usage or their runtime. The parts that required further considerations before implementing them are presented in the following.

### 6.1.1 Number of Available Hardware Threads

To determine the number of available hardware threads, the steady state distribution of a continuous time markov chain (CTMC) is used. As MATLAB offers no function to directly calculate this solution, a custom implementation was developed. It consists of two functions described in the following.

```
1 function solution = calulateAvailableHWTs(r_r, t_calc, n_hwt)
2 Q = (triu(ones(n_hwt+1)/t_calc,1) - triu(ones(n_hwt+1)/t_calc,2)) ...
```

```
3     * diag(n_hwt+1:-1:1) ...
4     + tril(ones(n_hwt+1)*r_r,-1) - tril(ones(n_hwt+1)*r_r,-2);
5   Q = Q - diag(sum(Q,2));
6   solution = solveCTMCsteadystate(Q);
7   end
```

This function is used to create the matrix $Q$, that describes the markov chain's transitions. In line 2 the upper minor diagonal is filled with the value $1/t_{\text{calc}}$. These values are scaled in line 3, since the rate of returns grows with more HWTs working at a given time. In line 4 the lower minor diagonal is filled with the value $r_r$. Afterwards, the main diagonal is constructed in line 5 by calculating the row sums of the matrix $Q$, that is only populated on the minor diagonal at this point. As described previously, the entries of the main diagonal of $Q$ have to be chosen for the rows of $Q$ to add up to 0.
Finally, the second function is called for deriving the steady state distribution of the CTMC described by $Q$.

```
1   function [solution] = solveCTMCsteadystate(Q)
2   solution = zeros(1, size(Q,1));
3   sum_vector = ones(1, size(Q,1));
4   lower_bound = zeros(1, size(Q,1));
5   upper_bound = ones(1, size(Q,1));
6   options = optimoptions('lsqlin','Algorithm','active-set','MaxIter',500);
7   solution = lsqlin(Q', solution, [], [], sum_vector, 1, lower_bound,
        upper_bound, [], options)';
8   end
```

This function implements equation (4.6) by using the least square solver provided by MATLAB. Lines 2 to 5 introduce auxiliary variables required for the solver, while line 6 sets the solver's options, such as the used algorithm and the maximum number of iterations to limit runtime. Finally, in line 7 the steady state solution in calculated. The solver is instructed to find a solution $\pi$, that meets three conditions:

1. $\pi Q = 0$
2. The entries in $\pi$ add up to 1
3. All entries in $\pi$ are between 0 and 1, since they represent probabilities

This vector, describing the probability of a certain number of HWTs being available, is then used to calculate the number of delegation messages that have to be sent.

### 6.1.2 Number of Delegation Messages

The number of delegation messages that are required for completely fulfilling a request is calculated by evaluating equations (4.28) and (4.29). They describe the probability of $n$ HWTM instances having $h$ or more HWTs available. For this purpose each possible distribution of available HWTs at the different management instances has to be considered. If a system, for example, consists of 100 managed HWTs and one management instance, there are 101 different states of the system. For two management instances there are already $51^2 = 2601$ different states, since each of the management instances can have between 0 and 50 HWTs available. For four management instances the number of states raises to $26^4 = 456976$.
Since the number of possible states grows rapidly, even for small quantities of HWTs and HWTMs, a naive implementation considering every single combination quickly exceeds the available memory and computing time. Besides, the program would run for hours on a desktop computer just for calculating a single value.
Therefore, a more sophisticated implementation was used. The HWTMs can be thought of as dice, each with $n_{\text{managed}}$ sides. The probability of rolling the side with $i$ eyes on it is $f_P(x = i)$, which is equal to the probability of a management instance to have $i$ HWTs available. The probability of $n$ HWTMs being able to provide $h$ or more HWTs then is equivalent to the probability of rolling $h$ or more eyes with $n$ of these dice. The outcome of a dice roll can be described by its generating function [Bar12], which in this case yields

$$f(z) = f_P(x = 0) + f_P(x = 1)z + f_P(x = 2)z^2 + \cdots + f_P(x = n_{\text{managed}})z^{n_{\text{managed}}} \quad (6.1)$$

and the outcome of rolling multiple dice is calculated by multiplying the function with itself. The probability of rolling $h$ eyes is equal to the coefficient of $z^h$. Applying this algorithm, the number of delegations is obtained by multiplying polynomials:

```
1 function res = prob_number_managers(n,h,p)
2
3 resultingPolynomial = 1;
4 for k = 1:n
5     resultingPolynomial = conv(resultingPolynomial, p);
6 end
7 res = sum(resultingPolynomial(h+1:end));
8
9 end
```

This function calculates the probability of $n$ managers being able to provide $h$ or more HWTs, if the probability of a single manager having $i$ HWTs available is $p(i)$. For this purpose the generating function characterized by `p` is multiplied with itself $n$ times in lines 4 to 6. In line 7 the probability of having $h$ to the maximum number of

HWTs available is calculated by summing up the individual probabilities. By using this algorithm, the calculations are finished nearly instantly and require almost no memory.
The rest of the formulas from section 4.5.2 are implemented as given without causing any problems.

### 6.1.3 Queueing Time

To calculate the queueing time for a request arriving in the queue of an HWTM the model of a M/M/1/N queue is used. In this calculation the two formulas (4.15) and (4.17) are causing numerical problems, when directly implementing them in MATLAB. Therefore, they will be treated separately in order to remove these numerical problems.

#### Numerical Problems When Calculating L

Equation (4.15) for the case of $a \neq 1$

$$L = \frac{a}{1-a} - \frac{(n_q+1)a^{n_q+1}}{1-a^{n_q+1}}$$

causes numerical instabilities, for large values of $n_q$. In this case the second fraction grows very large or extremely small, depending on the value of $a$. In the case of $a > 1$ the term $a^{n_q+1}$ grows very large. In this case the additional 1 in the denominator of the second fraction can be omitted, since it only has a negligible effect on the result of the calculation. Then the problematic terms cancel out and (4.15) simplifies to

$$L = \frac{a}{1-a} + (n_q+1). \tag{6.2}$$

In the other case of $a < 1$ the term $a^{n_q+1}$ is growing very small. Therefore, the second fraction in (4.15) can be omitted, since it is very close to 0. In this case, equation (4.15) simplifies to

$$L = \frac{a}{1-a}. \tag{6.3}$$

In summary, (4.15) is simplified to

$$L = \begin{cases} \frac{a}{1-a} & \text{if } a < 1 \\ \frac{a}{1-a} + (n_q + 1) & \text{if } a > 1 \\ \frac{n_q}{2} & \text{if } a = 1. \end{cases} \tag{6.4}$$

This solves the numerical issues for the calculation of $L$.

### Numerical Problems When Calculating $p_{n_q}$

The probability (4.17) of the waiting queue being filled completely causes numerical problems, as well. Unlike the other case no cancellation of the problematic parts can be reached. The problematic equation

$$p_{n_q} = \frac{(1-a)a^{n_q}}{1-a^{n_q+1}}$$

contains exponentials, as well, which causes problems, for large values of $n_q$.

In this case, the problems are be avoided by performing all calculations in logarithm. Using the logarithm's analytic properties, some simplifications can be applied to the equation:

$$\begin{aligned} \ln(p_{n_q}) &= \ln\left(\frac{(1-a)a^{n_q}}{1-a^{n_q+1}}\right) && (6.5) \\ &= \ln(1-a) + \ln(a^{n_q}) - \ln(1-a^{n_q+1}) && (6.6) \\ &= \ln(1-a) + n_q \ln(a) - \ln(1-a^{n_q+1}) && (6.7) \end{aligned}$$

Again, two cases have to be treated sparately. In the case of $a < 1$ the term $a^{n_q+1}$ can be neglected. Using the identity $\ln(1) = 0$ equation (6.7) simplifies to

$$\ln(p_{n_q}) = \ln(1-a) + n_q \ln(a). \tag{6.8}$$

In the case of $a > 1$ the term $a^{n_q+1}$ grows very large, making the 1 in the last term negligible. Therefore, equation (6.7) simplifies to

$$\ln(p_{n_q}) = \ln(1-a) + n_q \ln(a) - (n_q + 1)\ln(-a). \tag{6.9}$$

Combining these two cases, the logarithm of $p_{n_q}$ is given by:

$$\ln(p_{n_q}) = \begin{cases} \ln(1-a) + n_q \ln(a) & \text{if } a < 1 \\ \ln(1-a) + n_q \ln(a) - (n_q+1)\ln(-a) & \text{if } a > 1 \\ \ln(\frac{1}{n_q+1}) & \text{if } a = 1 \end{cases} \tag{6.10}$$

Afterwards $p_{n_q}$ is regained by reversing the logarithm function using an exponential function

$$p_{n_q} = \exp(\ln(p_{n_q})). \tag{6.11}$$

The two presented simplifications and transformations do not alter the results in a significant way but provide an algorithm that is capable of dealing with nearly arbitrary numbers without experiencing numerical problems.

Using the model presented in chapter 4 with the adaptions explained in this section, an simulation framework was implemented in MATLAB. To evaluate this model, an actual implementation of the presented HWTM was developed, that is presented in the following section.

## 6.2 Hardware Thread Manager

For evaluation purposes an implementation of the hardware thread manager (HWTM), that was presented in chapter 4, was developed as a part of the Many Threads Operating System (MyThOS) described in section 5.2. The implementation is done in C++ and it is encapsulated into a class `HWTManager`. The internals of this class are presented in this section. First the procedure used for setting up the HWTM infrastructure, that has to be done during the boot procedure of the operating system, is introduced in section 6.2.1. Then, the interface offered to application programs is described in section 6.2.2. Afterwards, the implementation of the main tasks of the HWTM, answering requests for new HWTs and returns of them, is outlined in sections 6.2.3 and 6.2.4.

### 6.2.1 Setup of the HWTM Infrastructure

When the operating system is booted, initially no HWTM instances are present and there is no inherent structure to the HWTs providing any information about the placement of the HWTM instances. The number of HWTM instances that should be initialized has to be specified at compile time. It results from the simulation of the performance model as described in 6.1 and is based on parameters of both hardware and application. To enable each HWT to contact its manager, the location and virtual address are stored in TLS. For providing the desired functionality, each HWTM instance has to maintain state information, that is summarized in the following. Afterwards, the initialization procedure setting up the HWTM infrastructure is presented.

#### State Representation

The state information that is maintained by every HWTM instance is grouped into two parts: Information about the locally managed HWTs and information about other HWTM instances.

- A HWTM instance needs a list of HWTs it is responsible for in order to return valid handles to these. Their IDs are stored in the array `size_t* list_managed_hwts`. The state of the locally managed HWTs is described using a boolean value, indicating if a HWT is occupied or available at the moment, which is stored in the array `bool* list_hwt_state`. An additional value denoting the index of the next HWT available is stored for quick access using the variable `long index_next_available_hwt`. It is set to $-1$, if no HWT is available at the moment.

- The assumed state of remote HWTM instances is stored in the array `size_t* list_remote_state`, that contains the last reported numbers of available HWTs at the other HWTM instances. To communicate with those instances, the IDs of their HWTs has to be known. Therefore, they are stored in the array `size_t* HWTM_locations`.

Additionally, the sizes of the arrays have to be known in order to avoid segmentation faults. All member variables have to be initialized at the startup of the system. This initialization and the placement of the HWTM instances is presented in the following.

#### Initialization

During the boot process, the operating system needs to call the method `static void setup(size_t num_hwt, size_t num_managers, size_t hwt_bsp)` to initialize the HWTM architecture. As parameters it has to specify the number of HWTs in the system, the number of HWTMs that should be initialized and the ID of the bootstrap processor (BSP), since it should be marked as occupied from the beginning because it is running application code. The initialization process is divided into several steps:

1. The HWT IDs are separated into intervals, depending on the number of HWTMs. The center of each interval is chosen as the location of a HWTM.

2. The number of managed HWTs per HWTM is calculated. This number may slightly differ among the HWTMs, if the number of HWTs is not divisible by the number of HWTMs.

3. For each HWTM instance a list of HWTs it is responsible for is created.

4. Using this information, each selected HWT is instructed to create a local HWTM instance.

5. Each HWT is notified of its responsible manager and its address. These values are stored in the corresponding TLS variables.

After completing this procedure the HWTM infrastructure is ready to be used by applications. The interface offered to them is specified in the next section.

### 6.2.2 Application Interface

The HWTM's main task is to provide the application program running inside the operating system with handles of hardware threads and taking them back for allocation to another part of the program. Therefore, it has to offer some interfaces to the application program for accomplishing these tasks. The API provided by the HWTM is specified as:

```
1 static HWTMRequestMessage* make_request_message(size_t amount);
2 static bool request_workers(HWTMRequestMessage* message);
3 static bool is_result_available(HWTMRequestMessage* message);
4 static void wait_for_result(HWTMRequestMessage* message);
5 static void return_worker(void* ec);
```

Each of these methods is declared static, such that the application does not need to know anything about the internal structure of the HWTM and the individual instances. The method `make_request_message` is used for creating a request message object, that encapsulates information like the requested amount of HWTs, the origin of the request and allows the application to retrieve the results. The structure of this request message object is covered later in this section.
A message created by calling this interface is used afterwards to request HWTs from the manager by calling the function `request_workers` and passing the message as the argument. The message is put into the waiting queue of the HWTM and the request is processed. The return value of the function indicates, whether the enqueueing of the request was successful. The application can issue additional requests or do other computations while the request is being processed. Using the method `is_result_available` it can check, whether the request is already done processing, which is a non-blocking operation, i.e. the method returns immediately independent of its result. Further, it is also possible for the application to wait for the result of the request by calling `wait_for_result`, which blocks and returns once the result is available.
The results of requests are handles to `ExecutionContext` objects, as described in section 5.2, which are passed as `void*`. These handles can be returned after use by passing them to the method `return_worker`. It is the application's responsibility to only use execution contexts it is currently in charge of and to stop using them after they are returned.
The only message an application has to be aware of is the request message and thus is detailed in the following.

### Message for Requesting HWTs

The messages used for requesting HWTs from the HWTM do not need to be initialized or altered by the application. All operations concerning these messages are implemented in the `HWTManager` class, while the actual message is a separate struct.

```
1 struct HWTMRequestMessage{
2   size_t origin;
3   size_t requested_amount;
4   int num_results;
5   void **result;
6   BitMutex* mutex;
7 };
```

A single message consists of a struct containing all relevant information: The origin (`origin`) of the request is required to return the response correctly and to internally

determine if a request was delegated or issued by the application. Together with the requested amount of HWTs (`requested_amount`) it provides the HWTM with all the information required for operation.
Since the number of HWTs returned may differ from the requested amount, the number of returned HWTs (`num_results`) is used to determine the size of the array of returned HWTs (`result`). The mutex contained in the message (`mutex`) is used for the waiting operations described previously in this section.

Exemplarily, a typical workflow of an application to acquire additional HWTs may look like this:

```
1 HWTMRequestMessage* message = HWTManager::make_request_message(20);
2 HWTManager::request_workers(message);
3 while (!HWTManager::is_result_available(message)){
4   //...Do some work...
5 }
6 for (size_t i = 0; i < message->num_results; i++){
7   void* worker = message->result[i];
8   //Assign a task to the worker
9 }
```

First a request for 20 HWTs is created and sent to the HWTM in line 1 and 2. Line 3 is used to wait for the result to be available. While waiting the application can perform different tasks. After the result becomes available the application iterates over the returned HWTs in line 6, acquires a handle to one of them in line 7 and then is able to assign a task to this HWT and thereby increase the degree of parallelism of the program.

### 6.2.3 Requesting Hardware Threads

When the method `request_workers` is called, as described in section 6.2.2, the request is forwarded to the responsible manager. This can be done quickly, because its HWT ID is stored as a TLS variable. The allocation of the requested HWTs is split up into three parts. First, locally available HWTs are assigned to the request. If the request is not fulfilled, the request is delegated to remote HWTM instances. Finally an answer to the request is sent to the requesting HWT. These steps are described separately in the following.

### Local Allocation

As a first step of the allocation locally managed HWTs are assigned to the request. This is accomplished by taking the next available HWT, marking it as occupied, creating an execution context and adding it to the result array and continuing with the next one. If the request can be fulfilled using the locally available HWTs, the function returns at this point. If it is not fulfilled after all locally available HWTs are assigned, delegation to other management instances is necessary, as introduced in section 4.5.1.

### Delegation of a Request

For the delegation of requests the basic algorithm proposed in section 4.5.1 is used. While HWTs are missing to fulfill the request, they are requested from remote instances, starting with the one that has the most HWTs available. For each of these delegation requests a new request message is created internally using the method `make_request_message`. This message is sent to the remote manager instance for further processing. All delegation messages are collected in an array. After all delegation messages are sent out the HWTM puts a task, a call to the function `process_delegation_results`, into its own queue to continue the processing of the current request at a later time, and continues with processing the next task from its queue.

The contacted HWTMs in turn perform local allocation, but do not delegate the request any further, as previously explained. The information, whether a request is a candidate for delegation is derived by comparing the origin of the request to the IDs of all available HWTMs.

When the method `process_delegation_results` is called, because delegation messages were sent, first it checks, whether all delegated requests were answered.
If responses are missing, the task is enqueued again for deferred processing. If all responses are available, they are collected into a single array containing both locally and remotely acquired HWTs. This array is passed to the `send_response` method that is also called, if no delegation is necessary at all and a request can be answered immediately.

#### Responding to the Request

After the requested amount of HWTs or all available ones have been assigned to a request it is passed to the `send_response` method. This method copies the collected handles to a new array of the correct size, if the amount of requested and returned HWTs is not identical. Afterwards the mutex contained in the request message is unlocked to enable the application to collect the requested resources.
Delegated requests are answered in the same way.

### 6.2.4 Returning a Hardware Thread

The application running on the operating system may use the acquires HWTs to perform tasks at its discretion. After it is finished using them it eventually returns them to the HWTM by passing their handles to the `return_worker` method. Its the responsibility of this method to make the correct HWTM mark this HWT as available again.
Since every HWT only knows the location and virtual address of its own HWTM, it cannot return HWTs directly that were acquired using delegation, because it does not know their managers location.
Therefore, a two-step algorithm was implemented. First a message is sent to the HWT that the application would like to return, instructing it to return itself. This HWT then sends a message its responsible HWTM to return itself.
Finally, when the message arrives at the correct HWTM, it marks the HWT as available, destroys the execution context representing this HWT and updates all other management instances with its new number of available HWTs.

# 7 Evaluation

The developed performance model is verified against measured values. The performance model is used to obtain theoretical values, as described in section 4, that are compared to experimental values measured from the implementation of the HWTM presented in section 6. An Xeon Phi running MyThOS, both detailed in chapter 5, is used as the evaluation environment.

To run the developed performance model, characteristics of both hardware and application have to be known. The determination of the required parameters of the utilized hardware and the HWTM architecture is detailed in section 7.1. To simulate applications with different parameters, a program running on MyThOS was developed to simulate an application and its requests. This simulation program is briefly described in section 7.2. Afterwards, the obtained results are presented in section 7.3 and discussed in section 7.4.

## 7.1 Hardware and HWTM Parameters

The individual parameters that characterize the hardware platform or the custom implementation of the HWTM are further listed and determined individually.

**$\mathbf{n_{hwt}}$** The parameter $n_{\text{hwt}}$ denotes the number of HWTs in the system. The Intel® Xeon Phi™ 5110P used for evaluation hosts 60 cores with 4 HWTs per core, which results in

$$n_{hwt} = 4\,\frac{\text{HWTs}}{\text{core}} \cdot 60\,\text{cores} = 240\,\text{HWTs}.$$

**$\mathbf{b_c}$** With $b_c$ he total bandwidth available for communication between two HWTs is denoted. Since communication in MyThOS is realized using queues located in main memory, the bandwidth available for communication with main memory

is used for this parameter. This bandwidth was measured in [FVS+13], where an average of

$$b_c = 120\,\mathrm{GB/s}$$

was determined for read and write operations.

**$t_{prop}$** By $t_{\mathrm{prop}}$ the average propagation delay of the communication infrastructure is denoted. Due to the usage of queues in main memory, the communication latency to main memory is considered here, which was experimentally determined in [FVS+13], as well. A communication latency of 302 cycles was measured. The Xeon Phi used for evaluation in this thesis runs at 1.053 GHz. This yields a communication latency of

$$t_{\mathrm{prop}} = \frac{302}{1.053\,\mathrm{GHz}} = 287\,\mathrm{ns}.$$

**$t_p$** The parameter $t_p$ describes the time, until a request for a single HWT is processed by the HWTM, assuming at least one HWT is available. Therefore, it depends on both the hardware platform and the the implementation of the HWTM. The time to processg a request for multiple HWTs was assumed to grow linearly with the number of requested HWTs. To verify this assumption the number of cycles fto process a request is measured for different request sizes using a single HWTM instance. Every request size is issued 1000 times. After each request the test program waits for the results, returns all HWTs and then waits for all returns to finish, before it issues the next request. The results are presented in figure 7.1. The assumption proves to be a good approximation, although the growth in runtime is slightly super-linear.

When performing a linear regression on the data, it was determined that requesting an additional HWT requires $1.2011 \times 10^5$ additional CPU cycles, which corresponds to

$$t_p = \frac{1.2011 \times 10^5}{1.053\,\mathrm{GHz}} = 114\,\mu\mathrm{s}.$$

**$t_{return}$** With $t_{\mathrm{return}}$, the time it takes for the HWTM to process the return of a single HWT is denoted. Since it depends on both the evaluation platform and the HWTM implementation, as well, it is also measured in MyThOS on the Xeon Phi. The application interface only allows returning one HWT at a time, so only this case has to be considered. The number of cycles for returning a HWT

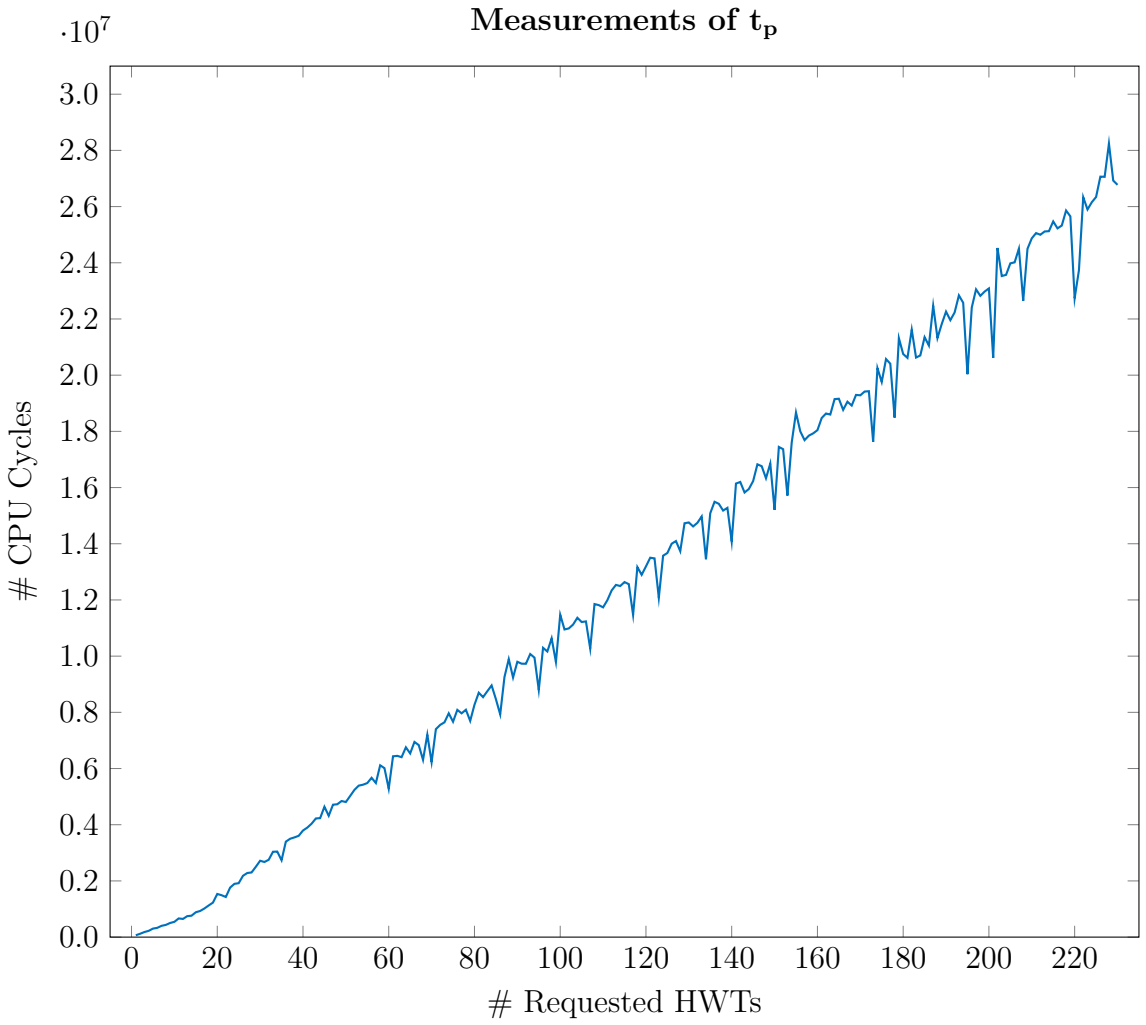



**Figure 7.1:** Measured values of $t_p$ for different amounts of requested HWTs

is measured to be $2.1795 \times 10^4$, which corresponds to a value of

$$t_{\text{return}} = \frac{2.1795 \times 10^4}{1.053\,\text{GHz}} = 20.7\,\mu\text{s}.$$

Using these values, response times for arbitrary application parameters are obtained from the MATLAB simulation. In the next section, the program used to simulate an application is outlined.

## 7.2 Simulation Program

In order to measure the actual response times of the HWTM infrastructure and compare them to theoretical values, a dummy application was developed that simulates the behavior of a real application while guaranteeing the following application parameters:

- Requests are issued at a fixed rate of $r_r$. This is accomplished by calculating the actual request rate and comparing it to the desired one. The request rate is calculated by dividing the number of issued requests by the total elapsed time. The application waits, if necessary, until the next request has to be issued.

- The size of each request is exactly $\bar{n}$.

- Each HWT is returned after $t_{\text{calc}}$. For this purpose, issued requests are cached and checked for completeness regularly. If a completed request is found, all HWTs it contains are instructed to wait for $t_{\text{calc}}$ before they return themselves.

To ensure a equal distribution of requests among the different HWTM instances, the origin of the requests is modified. After this modification it looks like the requests were sent from all HWTs in a round-robin manner. The HWTs running a HWTM instance are excluded from this mechanism. As well is the BSP, because it is running the dummy application and cannot be forced to wait and return itself.

A problem arises, if the request queue of the HWTM is already filled completely and the application wants to issue another request. In this case, the request is simply omitted to avoid a backlog and to ensure a constant request rate. If the request would be repeated after some time, the time for issuing this request would increase and the request rate might be higher afterwards, because the calculated request rate dropped due to the waiting operation.

Using this dummy application, three test cases are simulated and compared to the theoretical results in the next section.

## 7.3 Results

To evaluate the validity of the developed model, applications with different request rates, request sizes and calculation times are simulated. Three of those are presented

in this chapter. The parameters of these dummy applications are introduced in section 7.3.1 and the measured response times are compared with the theoretical ones in section 7.3.2.

### 7.3.1 Test Cases

To evaluate the validity of the performance model presented in chapter 4, the custom HWTM implementation is tested with different configurations of the dummy application that simulate the behavior of real applications. Three configurations are chosen to be presented in this thesis. The relevant parameters used for configuring the dummy applications are summarized in table 7.1.

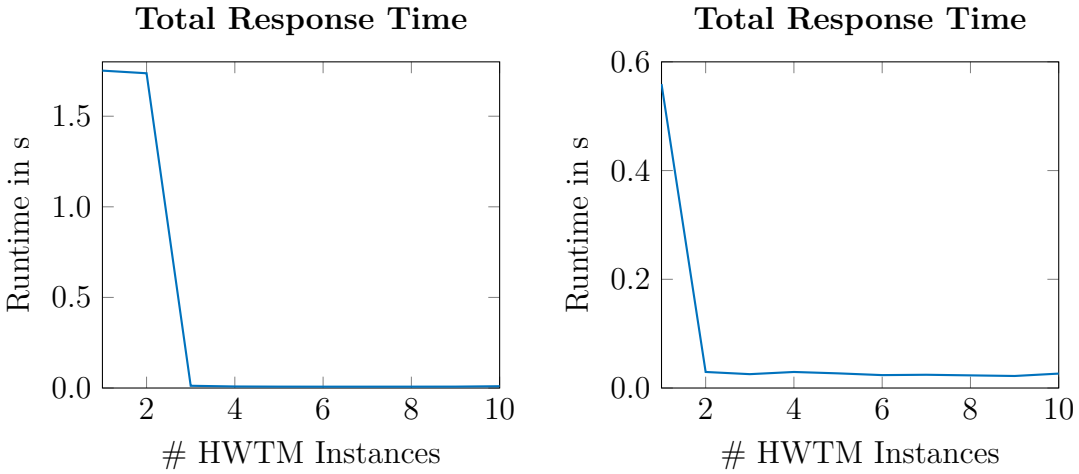
| Parameters | $r_r$ | $\bar{n}$ | $t_{\text{calc}}$ |
|---|---|---|---|
| P1 | 1000 | 20 | 1 ms |
| P2 | 100 | 30 | 10 ms |
| P3 | 5000 | 10 | 1 ms |

**Table 7.1:** Parameters used to configure the dummy application

The parameter set P1 characterizes an application that requests a moderate amount of HWTs once per millisecond. Every HWT then is assigned a small amount of work that takes 1 ms to compute.
A similar application is given with parameter set P3. There, the request rate is much higher, while the size of each request is only half as big. This configuration is expected to require more HWTM instances, since more requests have to be processed. Further, the number of delegation messages is expected to be lower, because a smaller request size is more likely to be fulfilled with less HWTMs.
With parameter set P2 an application issuing fewer requests is characterized. However, the request size is higher, as well as the calculation time until a HWT is returned. Therefore, it is expected to have a similar amount of HWTs available at any given time, but to require less HWTM instances to be performant due to the low request rate and the higher request size, that would cause lots of delegations when many HWTM instances are used.
Simulations are run for each of these parameter sets. Theoretical results from the performance model and experimental results of measurements in the HWTM implementation are presented in the next section.

### 7.3.2 Theoretical and Experimental Results

For each of the parameter sets introduced in the previous section, theoretical response times are obtained using the performance model. Experimental response times are measured using the implementation of the HWTM inside MyThOS running on a Xeon Phi.
The theoretical and experimental results for each of the parameter sets are presented in figures 7.2 to 7.4.

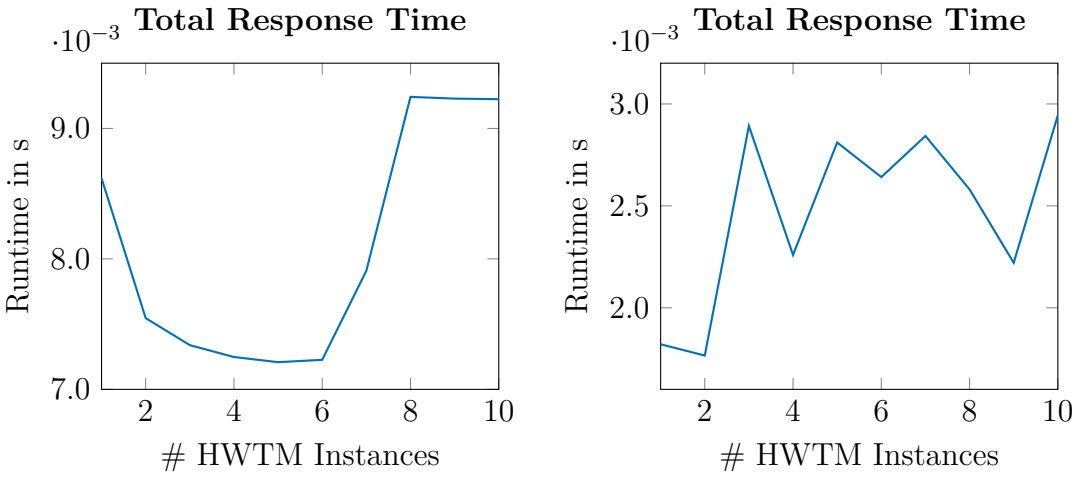



(a) Theoretical (b) Experimental

**Figure 7.2:** Theoretical and experimental response time of different amounts of HWTM instances for parameter set P1

The results for parameter set P1 are shown in figure 7.2. The performance model yields very high values of around 1.7 s for one and two HWTM instances. For more instances the response time drops to 123 ms and the settles to 80 ms. The model recommends to use four HWTM instances, cf. equation (4.36). The actual measured behavior of the measured runtime is a bit different as it starts at 560 ms for one HWTM instance and settles to approximately 260 ms for more instances. In regards of response time it would be best to use three HWTM instances in this case.

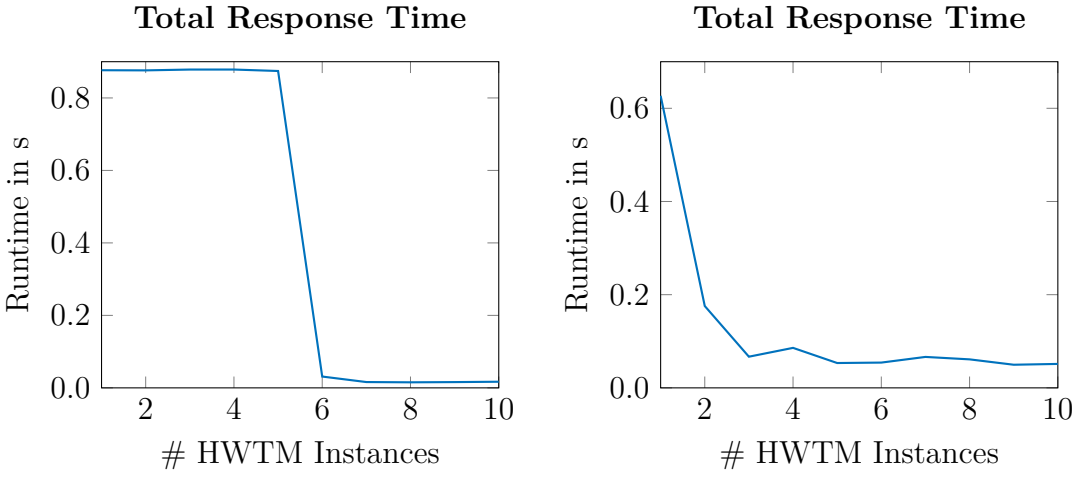
When using parameter set P2 the performance model predicts much lower response times, starting with 8.6 ms, as shown in figure 7.3a. The lower values result from the shorter queue lengths due to the lower request rate. The response time declines down to 7.2 ms for five HWTMs and then again raises up to 9.2 ms due to the increasing number of delegation messages. According to the performance model it is recommended to use five HWTMs to obtain optimal performance. The experimental results depicted in figure 7.3b show even lower response times of 1.8 ms and 1.7 ms




(a) Theoretical (b) Experimental

**Figure 7.3:** Theoretical and experimental response time of different amounts of HWTM instances for parameter set P2

for one and two management instances. For larger amounts of HWTMs the response time fluctuates around 2.5 ms. A reasonable choice would be to use two HWTM instances for this parameter set.




(a) Theoretical (b) Experimental

**Figure 7.4:** Theoretical and experimental response time of different amounts of HWTM instances for parameter set P3

For parameter set P3 response times of 870 ms are predicted by the performance model for up to five HWTM instances, as reflected in figure 7.4a. For more instances

the predicted response time drops to 16 ms. Figure 7.4b shows the experimental results. With only one HWTM instance it took 630 ms to receive a response. Afterwards, the response time decreases to about 60 ms. In this case, a reasonable choice would be to use three HWTM instances, while the performance model recommends using eight instances.

For the cases of parameters P1 and P3 the shape of the graphs for theoretical and experimental response times are similar. However, the response time seems to be overrated for low numbers of HWTMs and the drop from high to low response time occurs earlier in reality, than it is predicted by the performance model. For parameter set P2 the predicted decline from one to four HWTMs is only slightly visible in the experimental data. However, the predicted subsequent raise in response time is noticeable.

For all three cases it looks like the graph for the experimental results is a clinched version of the theoretical graph, i.e. the performance increases for lower numbers of HWTMs. Possible reasons for this are given in the next section.

## 7.4 Discussion

Although the theoretical and experimental response times do not match perfectly, they still are in the same order of magnitude. The current model seems to overrate the number of HWTM instances required to obtain certain response times, e.g. for parameter set P3 it predicts a drop in response time when using six instead of five instances, while the drop already occurs when using three instead of two.

The performance model incorporates the assumption of the representation of remote state being up-to-date all the time. However, a update message telling HWTM A that the number of HWTs at management instance B changed first has to pass the message queue of HWTM A. During this time A may decide to delegate a request to HWTM B, because it still believes that B has HWTs available, although this information is already outdated. Therefore, delegation messages may return too few HWTs in order to completely fulfill a request. The number of delegation messages may differ from the calculated values, because a HWTM cannot rely on its view of the remote state to be completely up-to-date.

Another problem arises from this outdated information and the fact that some requests are not fulfilled completely. The performance model assumes that a request for $n$ HWTs is assigned $n$ HWTs in all cases. However, this assumption may not

always hold. There are two reasons for that. First, there may be too few HWTs available in the overall system. Second, the outdated information about the remote state may lead to delegations to a HWTM that itself has too few HWTs available and therefore returns fewer than requested.
If one of these cases occurs and a request is not fulfilled completely, no execution contexts have to be instantiated for the HWTs that are requested but not returned. This dramatically reduces the processing time of this request and therefore the waiting time for all other requests in the queue.

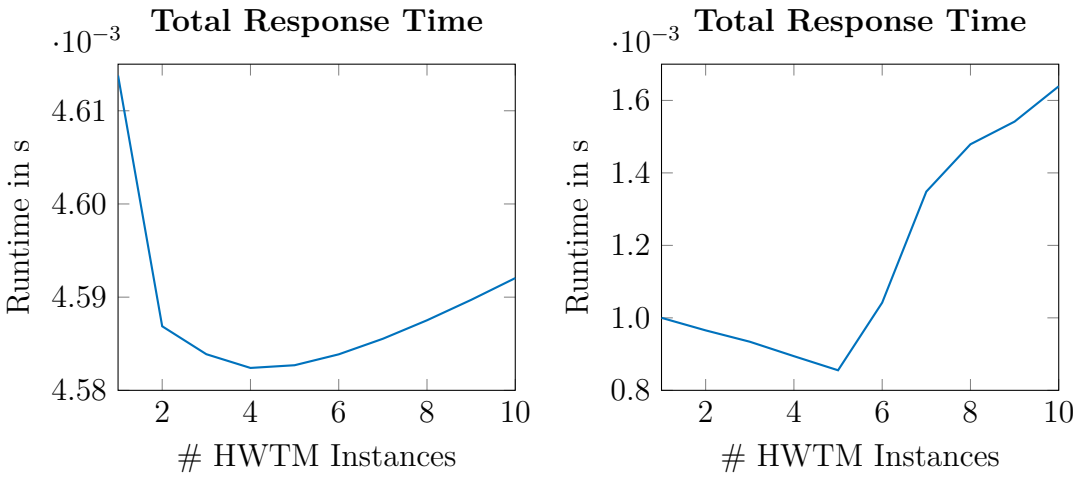
Yet another effect is caused by the outdated view of the remote state. If HWTM A has no HWTs available and it assumes that the other HWTMs has no HWTs available, as well, two effects add up. First, the local computation is much faster, as explained before. Second, HWTM A does not send out any delegation messages and therefore sends a reply to the request immediately after processing the request locally. In this case, there is no need to send additional messages, that occupy queues of remote managers, and to wait for the results of these delegation messages. Thereby the response time is massively reduced which again speeds up the processing of the queue and reduces the time a request spends waiting in this queue. Considering two HWTM instances A and B, where both have a few requests in their queues, an even worse scenario is possible. A event flow may look like this:

1. B tells A, it has 50 HWTs available.
2. A delegates a request for 50 HWTs to B and updates its local view. It now assumes B to have 0 HWTs available.
3. B processes a request for 2 HWTs from its queue.
4. B tells A, it has 48 HWTs available.
5. A delegates a request for 48 HWTs to B and updates its local view. It now assumes B to have 0 HWTs available.
6. B processes a request for 2 HWTs from its queue.
7. B tells A, it has 46 HWTs available.
8. and so on

Depending on the length of B's waiting queue there might be no HWTs available at the time the first delegation message from A is processed. The same applies for all

following delegation messages. Then again the processing time is reduced compared to the assumed one and the assumptions of the performance model are violated.

However, the fact that the theoretical and experimental response times are within the same order of magnitude and often close to each other proves the main assumptions of the performance model to be correct.




(a) Theoretical (b) Experimental

**Figure 7.5:** Theoretical and experimental response time of different amounts of HWTM instances without the need of delegation

Figure 7.5 shows a case, where no delegation is necessary. To achieve this a low request rate combined with a low request size is chosen. The request size has to be low enough to allow a single manager to completely fulfill the request, even if 10 HWTM instances are present. A request rate of $r_r = 10$ and a request size of $\bar{n} = 20$ is chosen. When comparing the experimental with the theoretical results, the general shape of the two graphs seems similar. When using the performance model a decreasing response time is observed for up to 4 HWTM instances, while the experimental results still show an improved performance when using 5 instead of 4 instances. These numbers are much closer to another, than in the cases presented in the last chapter, where delegation is necessary in all cases. Also, the impression of the theoretical graph being a stretched version of the experimental graph is absent in this scenario.
This further reinforces the hypothesis, that the deviations seen in the previous section originate from the use of delegation and some discrepancy between implementation and modeling of both management of remote state information and delegation messages.

Some additional deviations are caused by the fact, that the dummy application used for evaluation does not resemble workloads caused by real applications. The dummy application issues requests at a fixed rate and returns its acquired HWTs at a fixed rate, as well. Therefore, it issues all messages to HWTMs in a clocked manner, i.e. all HWTs from one request are returned at the same time. This may not model real workload sufficiently, since the duration of computations carried out by worker threads often depends on input data.

As assumed, the decentralized HWTM infrastructure offers increased performance to applications for various combinations of application parameters. Even for the relatively low amount of HWTs in the evaluation environment, this effect was observed. It can be assumed, that larger systems benefit even more from the decentralization of the management of computational resources.
The general course of the HWTM's response time for different amounts of management instances is described adequately by the performance model, that was developed in this thesis. Therefore, the model seems to represent the general operation of the HWTM.
However, the model overrates the response times and therefore calculates the optimal amount of management instances to be higher, than it is in reality.
Due to these results, the performance model's general structure seems to be valid, but it can be improved to better reflect the HWTM's actual operation.

# 8 Outlook

In this thesis, a management model for the management of hardware and software threads in a manycore environment is developed. The performance model presented in chapter 4 provides a possibility to compute the optimal amount of management instances for a decentralized and distributed management architecture. It describes the response time of this architecture depending on the management's degree of decentralization. An operating system based on a distributed microkernel and a manycore processor serve as a platform for the evaluation of the validity of the model, as introduced in chapter 5. Both performance model and management architecture are implemented as detailed in chapter 6. Several numerical issues are solved during the development process. This implementation is used to generate response times for several test cases. The performance model generates theoretical values for given hardware and software characteristics, while the implementation of the management architecture is used to measure experimental values. These values are compared and discussed in chapter 7.
The response times predicted by the developed performance model do not perfectly match the experimental results. It seems to overrate the response time in general. However, the general shapes of the graphs, showing the response times for different amounts of management instances, seem similar. The graph depicting the theoretical values appears to be a stretched version of the one showing experimental values.

A major issue the current performance model suffers from, is the fact that update messages are modeled to be processed immediately, but they are actually placed into the message queue and are processed like all other messages. Therefore, a HWTM instance has no real time view of the state of other HWTM instances. To improve the accuracy of the performance model a way of modeling this outdated information and making decisions based on this has to be found. One possibility to reduce the severity of the problem would be to include timestamps into update messages and only process the most recent update message for each HWTM instance. However, that would not completely solve the problem, since after issuing a delegation request, a HWTM still may process other requests before this delegation request. Therefore, no information about the state of the HWTM at the time a delegation request is processed can be provided. Attempts to find a way of doing this would result in the need of synchronizing operations between different HWTM instances and forcing

them to wait for another. This is completely contrary to the idea of distributing the processing of requests among different instances. Therefore, a model properly describing which information is available at what time is required.

The performance model in this thesis is based on the assumption of the request rate, request size and calculation time of each task being poisson distributed. While this assumption may be adequate for many applications, some others might show completely different characteristics. For example, an application could show both high and low request rates, that could more appropriately be characterized by a multimodal Gauss distribution. This could also be the case for the calculation time and the request size. The performance model in its current form is not able to deal with completely different distributions and cannot easily be adapted to deal with every possible input.
A possibility to deal with arbitrary input densities is to adapt the model to use the network calculus [LT04], that offers a way of characterizing completely arbitrary densities for all input variables. These variables do not even need to be identically distributed. Network calculus is based on ideas from system theory and it adopts these ideas to offer ways to calculate processing times of systems by characterizing both the densities of the input and the behavior and connections between the processing elements. However, adapting the performance model to employ the ideas of network calculus requires a major revision of the performance model, since all steps have to be transformed into network calculus.

Programs in HPC usually not only perform their computations on a single node, but rely on distributing it over several nodes to further improve their performance. The performance model in this thesis only considers a single processor on a single computing node. When extending it to multiple nodes, different communication latencies and bandwidths have to be incorporated, since inter-processor and inter-node connections show different characteristics than a typical interconnect within a single processor. When implementing such a system, every computing node might need one main HWTM instance, that communicates with the other nodes and then distributes work to other HWTMs within the same node, building a hierarchical structure. Another challenge could be to find a way of making reasonable delegation decisions based on the location of available HWTs. A delegation could only result in a performance improvement if HWTs on the local node are available, while delegating a request to a remote node may degrade performance. Furthermore, the application should be given the possibility to specify the locality of the requested HWTs. For some scenarios it might be necessary to only acquire HWTs located within the same node to provide a low communication delay among them.

Another possible improvement concerns application developers, that want to utilize the HWTM infrastructure. In its current implementation this infrastructure is set

up during the booting procedure of MyThOS. For this purpose, the application developer has to specify the characteristics of the application. These characteristics have to be either estimated or measured by the developer. However, this task could also be incorporated into the operating system, such that it benchmarks the application's characteristic after launching it and then sets up an appropriate HWTM infrastructure for it.
The set up process could be further improved by making it dynamic. Some applications might change their characteristic over time, e.g. they request HWTs at a low rate most of the time, but increase this rate based on either events from within the application or external events. To deliver the best performance in all cases, it would be desirable to have the HWTM infrastructure scale up and down itself. To achieve this, the operating system has to monitor the application and add additional HWTM instances or remove some based on the current behavior of the application. In order to make good estimates of the applications' current and future behavior, tracking methods like a Kalman filter could be implemented.

Currently, thread identifiers are not included, since HWT IDs are used as identifiers, because only one thread is assigned to each HWT and there are no context switches between different threads on the same HWT. If identifiers are desired, they could be implemented as an increasing counter for each management instance. Each instance could then be assigned a range of valid values. After using all these values, it needs to synchronize with the other management instances to get a new range of values.

# Bibliography


[All90] Allen, Arnold O.: *Probability, Statistics, and Queueing Theory with Computer Science Applications*. 2nd ed., Academic Press, Inc., 1990, pages 269–274.

[ATBS13] Anagnostopoulos, Iraklis; Tsoutsouras, Vasileios; Bartzas, Alexandros; and Soudris, Dimitrios: *Distributed run-time resource management for malleable applications on many-core platforms*. In: *Proceedings of the 50th Annual Design Automation Conference*, no. ii, page 1, 2013.

[Bar12] Bartlett, Padraic: *Lecture 7: Generating Functions*. In: *Math 1d*, 2012.

[BG00] Brun, Olivier and Garcia, Jean-Marie: *Analytical solution of finite capacity M / D /1 queues*. In: *Journal of Applied Probability*, volume 37, no. 4, pages 1092–1098, 2000.

[Cou09] Council on Competitiveness: *Boeing Catches a Lift with High Performance Computing*, 2009.

[CRFS07] Cheveresan, Razvan; Ramsay, Matt; Feucht, Chris; and Sharapov, Ilya: *Characteristics of Workloads Used in High Performance and Technical Computing*. In: *Proceedings of the 21st Annual International Conference on Supercomputing*, pages 73–82, 2007.

[CZ03] Carle, G and Zitterbart, M: *Protocols for High Speed Networks: 7th IFIP/IEEE International Workshop, PfHSN 2002, Berlin, Germany, April 22-24, 2002. Proceedings*. From the series *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003.

[Dow97] Downey, Allen B: *A model for speedup of parallel programs*. University of California, Berkeley, Computer Science Division, 1997.

[FKH08] Faruque, Mohammad Abdullah Al; Krist, Rudolf; and Henkel, Jörg: *ADAM: Run-time agent-based distributed application mapping for on-chip communication*. In: *Proceedings - Design Automation Conference*, pages 760–765. *DAC '08*, ACM, New York, NY, USA, 2008.

[FVS+13] Fang, Jianbin; Varbanescu, Ana Lucia; Sips, Henk; et al.: *An Empirical Study of Intel Xeon Phi*. In: *arXiv preprint arXiv:1310.5842*, 2013.

[GBG02] Garcia, Jean-Marie; Brun, Olivier; and Gauchard, David: *Transient analytical solution of M/D/1/ N queues*. In: *Journal of Applied Probability*, volume 39, no. 4, pages 853–864, 2002.

[HRY+08] Haibo, Silas Boyd-wickizer; Rong, Chen; Yandong, Chen; et al.: *Corey: An Operating System for Many Cores*. In: *Symposium A Quarterly Journal In Modern Foreign Literatures*, pages 43–57, 2008.

[HW10] Hager, Georg and Wellein, Gerhard: *Introduction to High Performance Computing for Scientists and Engineers*. From the series *Chapman & Hall/CRC Computational Science*. CRC Press, 2010.

[Int12] Intel: *Intel ® Xeon Phi ™ Coprocessor - the Architecture*, 2012.

[Int13] Intel: *Intel® Xeon Phi ™ Coprocessor System Software Developers Guide*, 2013.

[KBL11] Kobbe, Sebastian; Bauer, Lars; and Lohmann, Daniel: *DistRM: Distributed Resource Management for On-Chip Many-Core Systems*. In: *Proceeding CODES+ISSS '11 Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 119–128, 2011.

[LT04] Le Boudec, Jean-Yves and Thiran, Patrick: *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. In: *Online*, volume 2050, pages xix–274, 2004.

[MAT15] MATLAB: *version 8.5.0 (R2015a)*. The MathWorks Inc., Natick, Massachusetts, 2015.

[Med02] Medhi, Jyotiprasad: *Stochastic Models in Queueing Theory*. Elsevier Science, 2002.

[MSD+15] Meuer, Hans; Strohmaier, Erich; Dongarra, Jack; Simon, Horst; and Meuer, Martin: *TOP500 Supercomputer Sites*. 2015.

[Nor98] Norris, James: *Markov Chains*. Volume no. 2008 from the series *Cambridge Series in Statistical and Probabilistic Mathematics*. Cambridge University Press, 1998.

[OSK+11] Oechslein, Benjamin; Schedel, Jens; Kleinöder, Jürgen; et al.: *OctoPOS: A Parallel Operating System for Invasive Computing*. In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, pages 9–14, 2011.

[SD11] Severance, Charles and Dowd, Kevin: *High performance computing*. 2nd editon, O'Reilly, 2011.

[Tan01] Tanenbaum, Andrew S: *Modern Operating Systems, 2nd edition*, 2001.

[THH+11] Teich, Jürgen; Henkel, Jörg; Herkersdorf, Andreas; et al.: *Invasive Computing—An Overview*. In: Hübner, Michael and Becker, Jürgen (editors): *Multiprocessor System-on-Chip*, pages 241–268, Springer New York, 2011.

[ZSU+09] Zipf, Peter; Sassatelli, Gilles; Utlu, Nurten; et al.: *A Decentralised Task Mapping Approach for Homogeneous Multiprocessor Network-On-Chips*. In: *International Journal of Reconfigurable Computing*, volume 2009, pages 1–14, 2009.