Mechanisms for Energy-Efficient Processor Allocation and Redistribution on Manycore Systems

Von der Fakultät 1 - MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik der Brandenburgischen Technischen Universität Cottbus-Senftenberg genehmigte Dissertation zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Master of Science (M.Sc.)

Philipp Gypser

geboren am 10.07.1990 in Cottbus

Vorsitzender: Dr. M. Reichenbach
Gutachter: Prof. Dr. J. Nolte
Gutachter: Prof. Dr. W. Schröder-Preikschat
Gutachter: Prof. Dr. A. Polze

Tag der mündlichen Prüfung: 04.05.2023

DOI: https://doi.org/10.26127/BTUOpen-6385

Abstract

Multi- and manycore processors promise to combine high overall peak performance with moderate power consumption to meet the constantly growing demand for computational power under the energy constraints of today's CMOS technology. Future systems with manycore processors are expected to contain a huge amount of cores, which exceeds the number of processes that will run simultaneously. Consequently, processor time sharing approaches, that introduce significant overhead from regular context switches in common Operating Systems, will no longer be necessary.

This work investigates mechanisms for scalable and energy-efficient spatial partitioning of multi- and manycore processor systems. In addition, it explores the implications of exclusive processor core allocation to user processes due to the absence of temporal multiplexing and offers approaches to ease the adaptation to the new programming model.

The proposed mechanisms achieved fast thread allocation which motivates applications for dynamic thread allocation and benefits performance as well as energy efficiency. The efficiency control and resource revocation mechanisms detect and prevent wasteful and inefficient resource occupation from poorly optimized or malicious processes. In this way, the global efficiency of the system is optimized. The dynamic processing resource allocation and revocation handling has been integrated into a task parallel runtime system, to disburden the application programmer from manual implementation and to increase productivity.

Zusammenfassung

Mehrkern- und Vielkernprozessoren versprechen eine hohe Spitzenrechenleistung bei moderatem Stromverbrauch, um den ständig wachsenden Bedarf nach Rechenleistung mit den Energiebeschränkungen der heutigen CMOS-Technologie zu erfüllen. Es wird erwartet, dass zukünftige Systeme mit Vielkernprozessoren eine riesige Anzahl von Rechenkernen enthalten werden, welche die Anzahl der gleichzeitig laufenden Prozesse übersteigt. Folglich werden Ansätze für das zeitliche Multiplexing der Prozessoren, die in bestehenden Betriebssystemen erhebliche Kosten durch regelmäßige Kontextwechsel verursachen, nicht mehr zwingend erforderlich sein.

Diese Arbeit untersucht Mechanismen für eine skalierbare und energieeffiziente räumliche Partitionierung von Mehrkern- und Vielkernprozessorsystemen. Darüber hinaus werden die Auswirkungen der exklusiven Prozessorkernzuweisung, aufgrund des fehlenden zeitlichen Multiplexings, für den Benutzer untersucht und Ansätze zur Erleichterung der Anpassung an das neue Programmiermodell angeboten.

Die entwickelten Mechanismen erreichen eine schnelle Threadallokation, die Anwendungen zur dynamischen Ressourcenbelegung motiviert und damit sowohl die Rechenleistung als auch die Energieeffizienz erhöht. Die Mechanismen zur Effizienzkontrolle und zum Entzug von Ressourcen erkennen und verhindern eine verschwenderische und ineffiziente Ressourcenbelegung durch schlecht optimierte oder bösartige Prozesse. Auf diese Weise wird die globale Effizienz des Systems verbessert. Die dynamische Belegung von Rechenressourcen und die Behandlung von Ressourcelimitierungen wurden in ein aufgabenparalleles Laufzeitsystem integriert, um Anwendungsprogrammierer von der manuellen Implementierung zu entlasten und die Produktivität zu steigern.

Contents

1 Introduction 1

1.1 Problem Statement . . . 2

1.2 Main Contributions . . . 4

1.3 Structure of this Thesis . . . 4

2 Background 7

2.1 Multi and Manycore Architectures . . . 7

2.1.1 Core Composition . . . 8

2.1.2 Multithreading . . . 8

2.1.3 Tiling . . . 10

2.1.4 NoC Topology . . . 11

2.2 Dynamic Power Management in Multi- and Manycore Processors . . . 11

2.2.1 Energy Dissipation in CMOS Processors . . . 12

2.2.2 Dynamic Power Consumption Reduction Techniques . . . 12

2.2.3 Processor Power Management from the Operating System's Perspective 13

2.2.4 Dynamic Performance Boosting . . . 16

2.3 Operating System Processor Allocation . . . 16

2.3.1 Intra-Program Scalability . . . 17

2.3.2 Optimization Criteria . . . 19

2.3.3 Processor Multiplexing . . . 21

2.3.4 Spatial Processor Partitioning . . . 23

2.3.5 Application Profiling . . . 27

2.3.6 Linux . . . 30

2.3.7 iRTSS . . . 31

2.3.8 MyThOS . . . 34

2.4 Thread Management in Parallel Runtime Systems . . . 36

2.4.1 OpenMP . . . 36

2.4.2 Intel Threading Building Blocks . . . 36

2.5 Conclusion . . . 37

3 Energy-Efficient Processor Allocation on Multi- and Manycore Systems 39

3.1 Analogy to Memory Management . . . 39

3.1.1 Abstraction . . . 40

3.1.2 Virtualization 41
3.1.3 Replacement 42
3.1.4 Working Set 42
3.1.5 Fragmentation 43
3.1.6 Resource Quality 43
3.2 Requirements and Assumptions 43
3.2.1 Hardware Model 43
3.2.2 Application Model 44
3.2.3 Partitioning and Allocation Mechanism 44
3.2.4 Profiling and Redistribution 45
3.2.5 Flexible Task-Parallel Runtime System 46
3.3 Hierarchical Processor Allocation 46
3.3.1 Processor Topology Tree 46
3.3.2 Resource Management Approaches 47
3.3.3 Hierarchical Processor Pools 48
3.3.4 Pool Balancing Strategy 50
3.3.5 Placement Strategy 52
3.4 Application Profiling and Processor Redistribution 55
3.4.1 Online Application Profiling 56
3.4.2 Processor Revocation 60
3.5 Dynamic Processor Allocation in Task-based Parallel Runtime Systems 61
3.5.1 Dynamic Thread Allocation 61
3.5.2 Worker Suspension 63
3.5.3 Worker Resumption 65
3.6 Summary 67

4 Implementation **69**
4.1 Processor Allocator 69
4.1.1 User Interface 70
4.1.2 Architecture Overview 71
4.1.3 Thread Context Allocation 73
4.2 Application Profiling and Resource Redistribution 76
4.3 Dynamically Sized Worker Pools in Threading Building Blocks 77
4.3.1 Dynamic Worker Allocation 78
4.3.2 Worker Suspension and Resumption 79

5 Evaluation **81**
5.1 Measuring the Idle State Power Consumption and Wakeup Latency of a Real System . . . 81
5.1.1 Evaluation System . . . 82
5.1.2 Accurate Time Measurement on Multi-Core Processors . . . 83
5.1.3 Power Measurement using RAPL . . . 83
5.1.4 Experimental Measuring of Wakeup Latency . . . 84
5.1.5 Energy Consumption . . . 86
5.1.6 Conclusion . . . 87
5.2 Thread Allocation Latency . . . 88
5.2.1 Setup . . . 88
5.2.2 Expectations . . . 89
5.2.3 Results . . . 90
5.2.4 Conclusion . . . 90
5.3 Evaluation of the Energy Consumption and Execution Time on the Example of a Mandelbrot Set Rendering Application . . . 91
5.3.1 Mandelbrot Set Benchmark Application . . . 92
5.3.2 Setup . . . 93
5.3.3 Expectations . . . 94
5.3.4 Results . . . 95
5.3.5 Conclusion . . . 102
5.4 Automated Processing Resource Allocation in the Parallel Runtime System . 103
5.4.1 Setup . . . 103
5.4.2 Expectations . . . 105
5.4.3 Results . . . 105
5.4.4 Conclusion . . . 106
5.5 Impact of Dynamic Resource Allocation in Parallel Runtime Systems on Performance and Energy Efficiency . . . 108
5.5.1 Setup . . . 109
5.5.2 Expectations . . . 110
5.5.3 Results . . . 111
5.5.4 Conclusion . . . 114
5.6 Resource Efficiency Control through Online Application Profiling and Resource Revocation . . . 115
5.6.1 Setup . . . 115
5.6.2 Expectations . . . 117

5.6.3 Results 118
5.6.4 Conclusion 122
5.7 Optimizing Energy Efficiency using Dynamic Resource Redistribution 124
5.7.1 Setup 124
5.7.2 Expectations 125
5.7.3 Results 125
5.7.4 Conclusion 125
5.8 Summary 126

6 Conclusion **129**

A Energy Dissipation in CMOS Processors **133**
A.1 Static Power Consumption 133
A.2 Dynamic Power Consumption 134
A.2.1 Transient Power Consumption 134
A.2.2 Capacitive-Load Power Consumption 135

B CPU Instructions for Entering Idle States **137**
B.1 HLT (Halt) 137
B.2 MONITOR/MWAIT 137
B.3 Intel's UMONITOR/UMWAIT 138
B.4 AMD's MONIORX/MWAITX 138
B.5 PAUSE 138
B.6 TPAUSE 139
B.7 Conclusion 139

C Acronyms **141**

List of Figures **143**

List of Tables **147**

Bibliography **149**

CHAPTER 1

Introduction

For decades, the microprocessor industry has been committed to the continuous improvement of single-core processors by increasing clock frequency and core sophistication to maximize serial performance. Due to the enormous requirements in die area for more and more complex cores and constraints in power consumption, which are limited by the chip's heat density, the industry responded by halting increases in core sophistication and clock rate improvements. Instead, multiple processor cores are integrated into a single chip, producing a multi- or manycore processor to achieve higher overall peak performance without increasing power consumption and complexity of every single core. In this manner, the energy efficiency in form of computation per watt can be improved by scaling the energy consumption linearly with the number of cores instead of exponentially with the frequency and voltage. Thereby, the former trend of integrating a huge amount of simple in-order cores in a manycore processor went to a smaller but increasing number of more complex and powerful out-of-order cores. So, not only do today's large-scale supercomputers and cloud computers already contain tens to hundreds of cores per socket, but also mobile and Internet of Things (IoT) devices accommodate an increasing number of processor cores[54, 107]. However, future processors are expected to contain a much larger number of cores.

In order to take full advantage of the potential of a multi- or manycore processor, applications need to divide their work and employ multiple threads in parallel to process a problem cooperatively. Parallel runtime systems ease the development of parallel applications, because they implement the basic functionality of the parallel execution model, for example, taskification, synchronization, work distribution, and work balancing.

Usually, parallel programs operate in multiple phases with variations in parallel work that can be executed concurrently. So, a single process is usually not able to keep all cores of a manycore processor fully utilized to derive the highest potential benefit from the available hardware. That is why multiple programs have to be executed in parallel. Due to process isolation, the programs are functionally not affecting each other. To overlap input and output

latencies and phases with low parallelism, the number of threads of all processes must be greater than than the number of processor cores.

The management of the available processor cores as a hardware resource is a typical task of the Operating System (OS). The process scheduler, as a part of the OS, has to decide which process thread to run on which processor core at which time. In order to approach specific optimization goals like throughput, fairness, or response time, the scheduler regularly switches the execution between multiple threads on a core, when the number of threads exceeds the number of cores available. This is called *time sharing*[32].

1.1 Problem Statement

The development of multi- and manycore processors were driven by the constantly growing demand for computational power. In addition to acquisition costs, energy consumption is one of the major cost factors in the operation of computing systems. The maximum performance of processors is limited both by the amount of energy available and by the dissipation of power loss in the form of thermal energy. Therefore, users have an interest in obtaining the highest possible ratio of computing work per energy input. Due to the breakdown of *Dennard Scaling*[24], only fractions of the integrated circuits of those processors can be active at the full frequency at any given time without violating Thermal Design Power (TDP) constraints. *Dark Silicon*[33, 100] refers to the amount of circuitry that is powered off in order to meet the given power constraints. The Operating System-directed Configuration and Power Management (OSPM) is responsible for managing processor power consumption and systematically putting unused cores to sleep to save energy and boost active cores. Thereby, choosing the right idle sleep depth means a trade-off between energy savings and wake-up latency when needed again[18, 23].

Parallel applications try to improve their performance by using multiple threads solving problems cooperatively. Unfortunately, creating and starting a new thread is an expensive procedure when using existing OSs. Repeating this process every time a new task is produced results in significant performance degradation, which contradicts the reason for using multiple threads. In order to mitigate this performance loss, common parallel runtime systems try to reduce the number of threads to be created by building and maintaining static thread pools instead of dynamically creating and destroying them. Accordingly, threads are created and kept in the thread pool until they are needed. After finishing a task, they are returned to the thread pool to be used again later[37, 22]. This behavior hides the dynamics in the application

parallelism from the OS and, thus, complicates the resource and power management of the OS.

Today's OSs still rely on processes as an abstraction for running programs to simulate exclusive access to all processor resources. The processes abstraction was originally introduced in the single-core era to provide pseudo concurrent execution and overlap latency of input and output operations when there is only one processor available. Hence, each process perceives its own virtual processor that is realized using time-division multiplexing of the processor where it is quickly switching between processes, running each for tens to hundreds of milliseconds[96]. In today's systems, offering real hardware parallelism due to multiple processor cores, processes are still in use although they can now be executed fully concurrently. A process can include multiple software threads that have an individual control flow but share the same address space and might be executed in parallel as well. However, processes on common OSs are still agnostic to the processor utilization and, therefore, tend to create more threads than free cores available. This leads to oversubscription of the processor cores which is still handled using preemptive scheduling that requires regular context switches[115]. Those context switches do not come for free. The direct overhead of a process context switch includes entering the kernel mode, switching the kernel stack, saving the old thread state, loading the new data into the registers, switching the address space, and refreshing the Translation Lookaside Buffer (TLB). Additionally, indirect overhead arises due to missing hotness of the TLB and all levels of caches which slows down the execution of the new process after switching. While the direct overhead takes about tens of processor cycles, the indirect costs can be much higher but are heavily dependent on the hardware platform, application behavior, and data set size[72, 99]. Hence, frequent context switches introduce significant overhead to the actual productive work.

Research OSs with a focus on manycore processors propose to follow the *one-thread-per-core execution model*[67, 86], where a multi-threaded process can allocate multiple cores, but a core is always allocated exclusively to only a single process. So, frequent context switches, including their overheads, are avoided and scheduling complexity can be reduced which benefits the scalability of the scheduling algorithm. Since the overall number of threads of all processes becomes limited to the number of cores in the system, thread allocation requests might fail, which requires all applications to act resource-aware. However, these research OSs do not focus on energy efficiency and thermal interdependency between all cores of the same processor socket. Furthermore, they lack dark-silicon management and require offline performance analyzes or specialized hardware for application profiling in order to rebalance resource allocation. Preemptive resource revocation is not supported.

When following the one-thread-per-core model, keeping temporally unused threads in pools blocks the associated cores from doing any productive work. Hence, common parallel runtime systems are not suitable for such resource-aware environments, because they do neither dynamically express the applications parallelism profile to OS nor are they able to deal with limited resources or resource revocation.

1.2 Main Contributions

This thesis made several research contributions in the field of energy-efficient resource management on multi- and manycore processor systems. It introduces a scalable processing resource management mechanism that improves energy efficiency compared to Linux. Therefore, it follows the one-thread-per-core execution model to avoid the overhead of regular context switches and systematically puts unused cores, considering the actual hardware topology, into a specific sleep state to reduce energy consumption and accelerate active cores. Additionally, the core allocation latency is reduced to motivate the runtime system to pass the application's dynamics in parallelism using dynamic core allocation and deallocated instead of thread pools.

This thesis introduces a resource revocation mechanism allowing asynchronous deprivation of cores that are already allocated to processes but do not meet the global requirements for energy efficiency with their utilization. As a basis for the dynamic resource balancing decisions, an online application profiling mechanism, that uses hardware performance counter and a proof-of-concept balancing strategy are introduced as well.

State-of-the-art parallel runtime systems are neither able to handle limited resources nor resource revocation. Hence, a concept for adapting existing parallel runtime systems to behave malleable and handle resource revocation is introduced and exemplarily integrated into Intel Threading Building Blocks (TBB).

1.3 Structure of this Thesis

The remainder of this dissertation is outlined as follows. The second chapter presents an overview of the architecture and properties of multi- and manycore processors and how their power consumption can be controlled by the OS. Additionally, the processor resource management of existing OSs and parallel runtime systems are examined. In Chapter 3,

requirements and design aspects for the scalable and energy-efficient organization of processing resources in the OS, mechanisms for core allocation and redistribution based on application profiling, and resource-aware thread management for task-based parallel runtime systems are discussed and concepts derived. Chapter 4 presents the integration of the developed resource management mechanism and application profiling into Many Threads Operating System (MyThOS) as well as the adaptation of a parallel runtime system in order to meet the requirements of malleable applications on the example of TBB. The benefits of resource-aware applications combined with hierarchical processor management are evaluated and the limitations are critically examined in Chapter 5. Chapter 6 summarizes the contributions of this thesis and gives an outlook on future work.

CHAPTER 2

Background

The goal of this thesis is to investigate scalable and energy-efficient processor resource management mechanisms for multi- and manycore processor systems. Therefore, this chapter examines the architecture and power management infrastructure of existing multi- and manycore processors. After that, processor resource management mechanisms of state-of-the-art OSs and approaches for online application profiling are analyzed. Finally, thread management in common parallel runtime systems is presented.

2.1 Multi and Manycore Architectures

Due to limits in power density, the microprocessor industry has stopped trying to just steadily increase single-core performance. Instead, multiple processor cores are integrated into a single chip to improve the overall performance without increasing the power consumption. Multi-core processors correspond to this approach but do only contain a few but complex cores. They still provide a high single-core performance so that a good performance can be gained even with moderate parallelism of the application. Manycore processors do not only contain more, but generally simpler cores and aim for power efficiency. The lower single-core performance needs to be compensated with massive parallelism. However, a clear distinction between multi- and manycore is difficult to make for recent processors because they include a high number of complex cores with good single-core performance. Nevertheless, there are multiple properties that allow a further classification of multi- and manycore processors. The following section examines the difference based on the core composition, multithreading support, Tiling, and Network on a Chip (NoC) topology. Multiple processor examples and their classifications are given in table 2.1.

Name	Cores	Composition	Multithreading	Tiling	NoC topology	Release
AMD EPYC 7003	64	homogeneous	2 way SMT	eight cores per tile	star[1]	2021
Intel Xeon Gold 6200	28	homogeneous	2 way SMT	one core per tile	2D mesh	2020
Intel Xeon Phi 7200	72	homogeneous	4 way SMT	two cores per tile	2D mesh	2016
Intel Xeon Phi 7100	61	homogeneous	4 way SMT	one core per tile	bi-directional ring	2013
Mellanox TILE-Gx72	72	homogeneous	-	one core per tile	2D mesh	2013
UltraSPARC T1	8	homogeneous	4 way interleaved	one core per tile	crossbar	2005

Table 2.1: Examples for multi- and manycore processors[20, 21, 56, 58, 62, 69, 102]

2.1.1 Core Composition

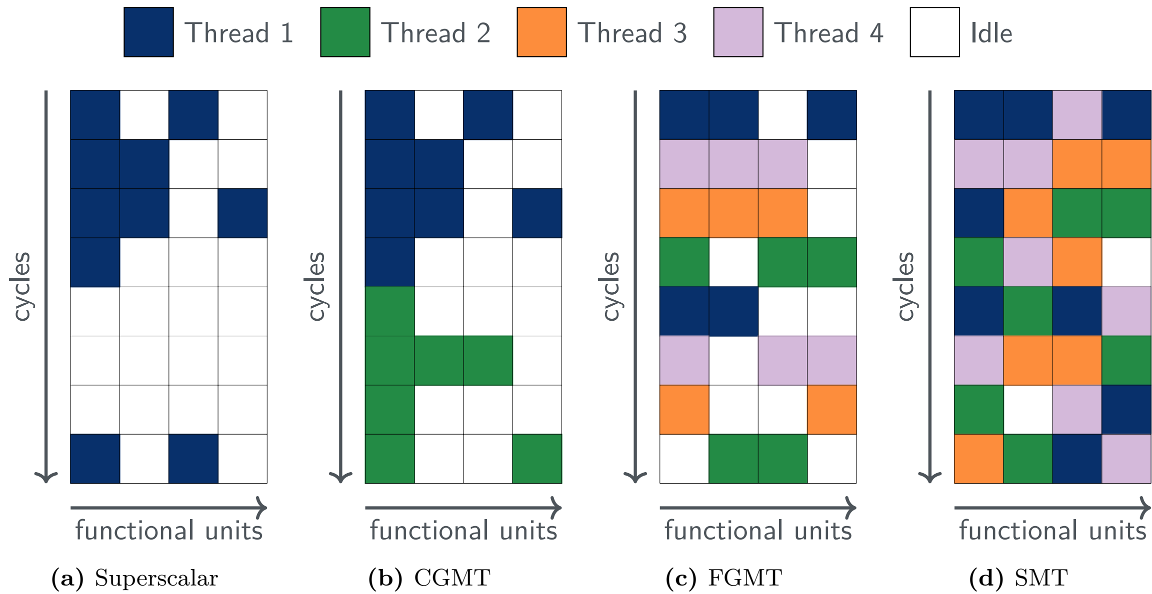
Multi- and manycore processors can be categorized by the composition of processing cores[77]. *Homogeneous* processors include a collection of identical cores. This simplifies hardware design, verification, and implementation as well as software development. Due to the end of *Dennard scaling*[24], that stated that the energy consumption per chip area stays constant even if the transistors get smaller and the number of transistors per area increases, the power density became the limiting factor for processor performance. This causes the *dark silicon effect*[33] so that not all transistors can be used at the same time and some areas of the chip remain dark. To counteract this effect, *heterogenous* processors integrate a mix of processing cores that differ in the power and performance characteristics or functionality. So, a task can be executed on the most suitable processor core with comparatively high performance and low energy consumption.

2.1.2 Multithreading

Hardware multithreading is the capability of a compute core to process multiple threads of execution concurrently. There are different types of hardware multithreading: *Temporal multithreading* and *Simultaneous Multithreading (SMT)*[12, 69, 93, 98]. These and their subtypes are explained in the following section and are illustrated in figure 2.1 for comparison.

Temporal Multithreading In temporal multithreading only instructions of one thread can be executed at a time and the processor core regularly switches between multiple threads. Hence, it is also called time-sliced or vertical multithreading. It is used to improve throughput

[1] Up to eight compute cores form a Core Complex (CCX) and share one last level cache. Each CCX is contained within a single die, called Core Complex Die (CCD). The individual CCDs are directly connected to the central I/O Die (IOD) over Global Memory Interconnect (GMI) and *Infinity Fabric*. The IOD connects all dies and contains the memory controllers as well as other Input and Output (I/O) devices[56, 58].



(a) Superscalar **(b)** CGMT **(c)** FGMT **(d)** SMT

Figure 2.1: Hardware multithreading types

by hiding memory latency. There are two variations of temporal multithreading. In the case of *Coarse-grain Multithreading (CGMT)*, a thread has full use of the core resources until a long-latency event like a memory stall occurs. In this case, the core switches to another thread in order to bridge the latency. This introduces overhead for the context switch that requires a flush of the instruction pipeline. Therefore, a context switch is only performed when the event is expected to exceed a certain latency. *Fine-grain Multithreading (FGMT)* (also called interleaved multithreading) switches between threads with a more fine granularity which happens typically at an instruction cycle boundary. A selection policy allocates the processor resources to threads that are currently ready and therefore not blocked due to memory stalls. In order to minimize switching costs, processors with fine-grain multithreading include special logic for thread switching.

Simultaneous Multithreading SMT schedules instructions from multiple threads on different functional units of a processor at the same cycle and is also called horizontal multithreading. It was introduced to improve the utilization of shared functional units in superscalar processors. However, SMT is also used to hide memory latency and increase throughput as well as energy efficiency. SMT has additional hardware costs compared to interleaved multithreading, because each pipeline stage has to track the corresponding thread identifier. The register set for the architectural state is duplicated for each individual thread, but the duplication of other components is processor-specific. Depending on the processor design and application,

SMT might decrease the performance if shared resources like Floating Point Unit (FPU), TLB, or caches become bottlenecks.

2.1.3 Tiling

Processors can be clustered into tiles where each tile contains a set of cores, a local cache that is accessed via a shared bus, and one NoC router or switch to communicate with other tiles. An example is given in figure 2.2.

Putting all cores of a multi- or manycore processor into a single tile promises a simple hardware implementation because a NoC is not needed but suffers from limited scalability of the shared memory bus. In contrast, putting each core into an individual tile requires the most additional die area for NoC interconnection but is most flexible and scalable in the number of cores that can be attached by just adding more tiles to the NoC[101, 114].

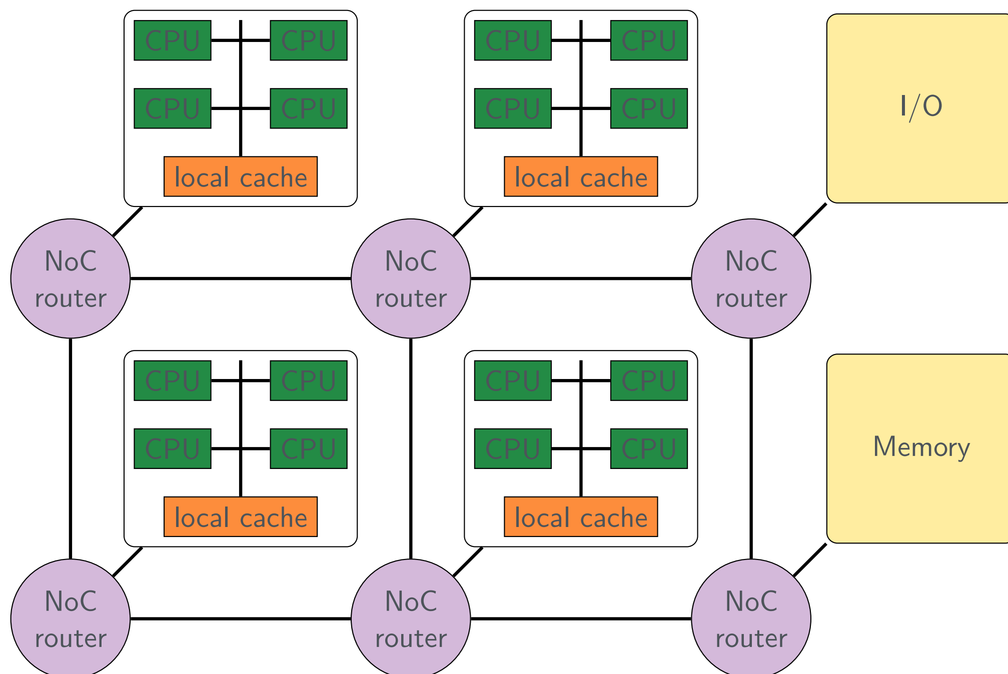


Figure 2.2: Example of processor tiling architecture with four tiles containing four cores and a local cache each. All tiles, memory, and IO devices are connected via NoC

2.1.4 NoC Topology

Multi- and manycore processors can also be divided by the underlying Network on a Chip (NoC) topology that connects the tiles, memory, and I/O devices. Thereby, they form the foundation for communication between those individual components while each topology has its own characteristics. Choosing a NoC topology means making a trade-off to meet the chip area, complexity, latency, scalability, and power constraints. As depicted in figure 2.3, common topologies are, for example mesh, ring, star, bus, and tree.

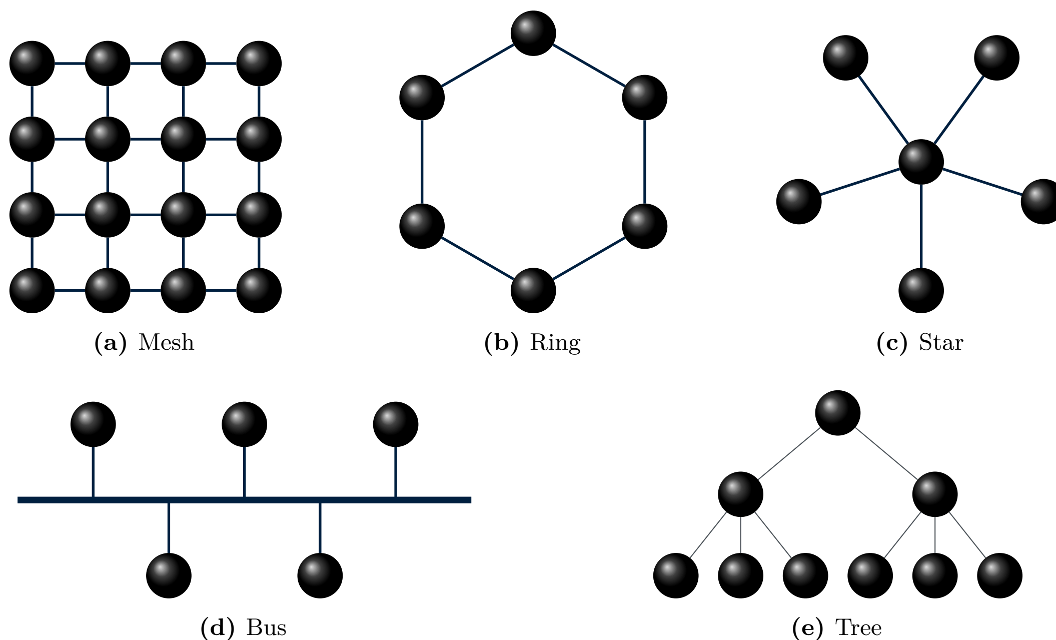


Figure 2.3: Examples of Network on a Chip topologies

2.2 Dynamic Power Management in Multi- and Manycore Processors

Multi- and manycore processors increase the overall peak performance and consume less energy than systems in which each processor has its own physical chip. Due to *dark silicon*, not all circuitry of those processors can be permanently active at the full frequency and the performance is therefore tightly coupled to the available energy budget. Hence, this section examines the sources of power dissipation and the capabilities to regulate it during runtime.

2.2.1 Energy Dissipation in CMOS Processors

Energy consumption is a significant criterion when designing integrated circuits. It is limited by the capabilities of the power supply and current requirements as well as the cooling capacity of the heat sink. When examining the energy consumption of processors, one has to think about the energy consumption of integrated circuits using Complementary Metal Oxid Semiconductor (CMOS) technology, because it has been the standard technology for the last five decades[111].

The total power consumption in CMOS technology arises from several sources. Therefore, it can be divided into static and dynamic components[61]. Static power consumption occurs when all inputs of a circuit are held at some constant level and is caused by leakage current due to parasitic diodes in the CMOS circuit. It increases with the supply voltage. Dynamic power consumption only occurs when switching between logical states in the circuit and consists of *transient power consumption* and *capacitive-load power consumption*. It contributes a major portion of the overall power consumption and scales linearly with the switching frequency and the supply voltage squared. Hence, performance improvements by frequency increase, and consequently an increase of the supply voltage as well in order to meet timing requirements, scale sublinearly to the power dissipation. Further explanation can be found in the appendix A.

2.2.2 Dynamic Power Consumption Reduction Techniques

In order to understand how the operating system can optimize the processor's energy consumption, this section examines existing mechanisms to dynamically influence the power dissipation during runtime[82].

Power Gating The most effective method for reducing power dissipation is to shut down currently unused blocks of the circuit by switching off the power supply. This technique is called *power gating* and prevents not only dynamic but also the static power dissipation, because it inhibits leakage current of unused but powered circuits. Although the benefits of high energy savings are promising, this technique requires additional logic to the circuit for power management and might cause increased time delays for entering and exiting power-saving states.

Clock Gating A second power reduction technique is *clock gating*. In contrast to *power gating*, it only avoids dynamic power dissipation by pruning the clock tree which prevents state transition in a specific block of the circuit. The leakage current of static power consumption remains and it requires additional logic for controlling the clock tree in the circuit.

Dynamic Frequency and Voltage Scaling The dynamic power consumption of CMOS-based processors increases quadratically with the supply voltage and linearly with the clock frequency. Instead of completely disabling a functional unit in the circuit, the power dissipation can be regulated by dynamically adjusting the clock frequency and supply voltage. Lowering the supply voltage (*undervolting*) promises the greatest energy savings but increases the time required to charge and discharge the capacitances in the circuit leading to slower operations. Therefore, the clock frequency needs to be adjusted to match the supply voltage in order to guarantee timing requirements. Decreasing the clock frequency (*underclocking*) directly reduces the dynamic power consumption linearly and also enables *undervolting* by being able to tolerate extended operation delays. When the maximum computing power is required, the opposite operation is also possible. Therefore, as long as the energy and cooling budget allow it, the clock frequency (*overclocking*) and supply voltage (*overvolting*) can temporarily be increased to boost the execution of the processor. *Dynamic voltage and frequency scaling* requires sophisticated hardware support which increases a processor's complexity. Additionally, the voltage and clock frequency will only be adjusted with a certain delay to fluctuating performance requirements.

2.2.3 Processor Power Management from the Operating System's Perspective

Today's CMOS-based processors offer mechanisms to adjust the power consumption according to the current performance requirements. However, the processor lacks knowledge about the process schedule and performance, and energy requirements of the user. This knowledge is essential when deciding whether or not to enter a specific power optimization state because state transitions introduce extra delay and consume energy by themselves as well. The operating system is responsible for process scheduling and is therefore aware of the near-future processor utilization. In addition, it might include own power management strategies and accept performance hints from the user. For this reason, the power reduction mechanisms of the processor are partly made controllable by software, especially the operating system. Modern processors offer multiple interfaces for operating the power management hardware. This section provides a brief overview of the most common ones.

Performance-States (P-States) Modern Central Processing Units (CPUs) include mechanisms to monitor thermal conditions and control power consumption. P-states [18, 57, 59] are operational performance states that allow switching between multiple voltage-frequency pairs while the processor is actively executing instructions. They are defined as performance states in the Advanced Configuration and Power Interface (ACPI)[60]. The performance increases with frequency and power consumption which is realized as frequency and voltage scaling in the processor's CMOS circuits.

P-states can be changed in up to 16 P-states from *P0* to *P15* where *P0* is the highest power/performance state which enables the highest possible frequency and each ascending P-state number represents lower power, lower performance state. The operating system controls P-state per core by accessing specific control, status, and limit registers in order to reduce the peak thermal load and save power. Hardware may limit P-states due to interdependencies between cores that affect the P-state (e.g. thermal constraints). Latest Intel CPUs offer support for *Hardware-Controlled Performance States (HWP)*. This technology is referred to as *Intel Speed Step* or *Intel Speed Shift*[18, 20]. When activated, the operating system is allowed to give hints about performance preferences, but the actual P-state control is performed by the hardware autonomously.

CPU Idle States (C-States) C-states [18, 23] are, unlike P-states, idle power-saving states which are used to shut down parts of the processor when unused. They are typically implemented using *clock-gating* and *power-gating* of individual functional units of the processor.

ACPI[60] defines a set of four logical C-states *C0* to *C3* where a higher number refers to a deeper sleep state with lower power consumption and potentially higher wakeup latency. Those logical sleep states do not necessarily refer to hardware C-states because the mapping is processor-specific. From a hardware perspective, we differentiate two types of C-states: core level *CC-states* and package level *PC-states*. Core level *CC-states* are used to shut down parts of individual cores. Only the CC-states can directly be influenced by the operating system. Entering those idle states can either be done using the ACPI interface or using a set of hardware specific instructions. An overview about CPU instructions for entering idle states is provided in appendix B. Each processor core might contain multiple hardware threads. Therefore, the CC-state of a core equals the lowest (not the deepest) C-state of all threads on this core. The number of available CC-states is hardware specific, but recent Intel processors specify idle states from *CC0* to *CC10* where some include substates[17]:

CC0 is the active state where the core is executing code at normal frequency and thus not an actual idle power-saving state.
CC1/CC1E is the least deep idle state leading to the lowest energy savings and wakeup latency. The core is halted and most clocks are stopped. Substate *CC1E* allows the package to throttle the core's frequency and voltage.
CC2 is only a temporary intermediate state before entering deeper sleep states.
CC3 flushes the core local first-level data and instruction cache as well as the second-level cache to the shared Last-Level Cache (LLC).
CC4 and CC5 are only temporary intermediate states before entering deeper sleep states.
CC6 saves the cores state to a dedicated Static Random-Access Memory (SRAM) and power-gates the core after completion. The core state will be automatically restored when exiting the sleep state.
CC7 - CC10 behave similar to *CC6*, but allow the package to enter a deeper *PC-state*

Package level C-states (*PC-states*) are able to shut down the circuits that support and connect the individual cores. The PC-state is limited to the lowest CC-state among all cores it contains and cannot explicitly be requested by the operating system. In order to enter a *PC-state* higher than *PC0*, all hardware threads and cores must be in an idle state. Intel mentions package level sleep states from *PC0* to *PC10* but not all states are defined:

PC0 is the active state where at least one core is executing code or did not enter a *CC-state* higher than *CC1*. In addition, the platform configuration might prohibit entering a package level low-power state.
PC1/PC1E is the least deep package idle state. In *PC1*, no additional power reduction actions are initiated. In substate *PC1E* throttles the cores frequency and voltage to a minimum.
PC2 is entered if all cores have requested *CC3* or deeper idle states, but constraints like a programmed timer in the near future or outstanding memory requests prevent the package from entering a deeper *PC-state*.
PC3 might flush and power of the LLC. Most Uncore clocks are stopped and Uncore voltages are reduced.
PC6 saves the core states before they are shut down and the Phase-Locked Loop (PLL) is turned off.
PC7 behaves similar to *PC6* but the LLC might get flushed.
PC8 equals *CC7* but the LLC must be flushed and powered off.

PC9 equals *CC8* but most uncore devices, except input and output devices, are power-gated.
PC10 equals *CC9* but the voltage regulators are in an optimized state (low power mode).

2.2.4 Dynamic Performance Boosting

The dynamic usage of performance states and idle states allows to build up headroom in power in the form of saved energy and thermal capacity. The processor monitors its activity and estimates the power consumption. If the power consumption is below the internally defined capacity of the hardware, it can be used to increase the performance in means of voltage and frequency of single cores for short periods of time without exceeding design limits. Examples for this technology are *AMD Core Performance Boost*[57] and *Intel Turbo Boost*[18]

2.3 Operating System Processor Allocation

The available processors of a computer system must be made accessible for user applications in order to do useful work. This is one type of resource management for which the OS is responsible. Therefore, it needs to fulfill two requirements. At first, it must provide an abstract representation of the actual hardware to increase the programmability and portability of the user application and, secondly, it must control the allocation of system resources to ensure the correct execution of the applications while maximizing the efficiency of the overall system. In order to take advantage of the high peak performance of multi- and manycore processors, *multiprogramming* has to be applied, because a single sequential application is not able to utilize numerous cores simultaneously. Multiprogramming can either be realized using *inter-program* parallelism, *intra-program* parallelism, or a combination of both. Inter-program parallelism denotes the parallel execution of multiple sequential applications, whereas intra-program parallelism terms the sequential execution of parallel programs[45]. This work applies a combination of both to meet the increasing number of available cores of future processors. The allocation of processing resources to applications depends on many criteria which are discussed in the following. Afterwards, examples for processing resource management in existing OSs are investigated.

2.3.1 Intra-Program Scalability

Intra-program parallelism as a kind of multiprogramming is one way to utilize the high parallel peak performance of multi- and manycore processors. The number of processing elements in future processors is expected to further increase, but parallel applications are possibly not able to benefit from even more resources[54]. Hence, when assuming an infinite amount of processing resources, what would be the optimal number of processors for an application? This question is investigated in the following.

Amdahl's Law In 1967, Gene Amdahl[3] emphasized that single computer systems reached their limits in computational power and that, in order to significantly accelerate the execution of applications to real problems, multiple computers need to be connected to permit a cooperative solution. He also noted that the speedup in the runtime of these applications does not scale linearly with the number of compute nodes used and concluded that the speedup is limited by the sequential part of an application.

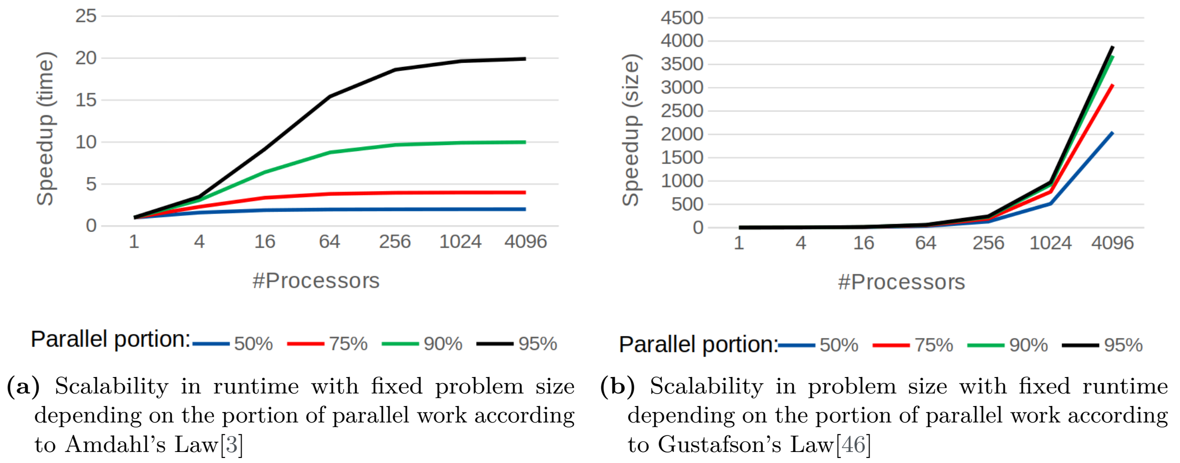
According to Amdahl, an application with a fixed problem size can always be divided into a parallel part ($P_{parallel}$) and a sequential part ($P_{sequential}$), so that applies $P_{parallel} + P_{sequential} = 1$. Therefore, the total runtime T equals the sum of the runtime of the sequential part $T_{sequential}$ and the runtime of the parallel part $T_{parallel}$. The speedup of the parallel part ($P_{parallel}$) is assumed to scale linearly with a number of processing units (n) applied to it while the time required to execute the sequential part ($P_{sequential}$) remains constant. Based on this assumption, a model for the expected application speedup depending on the parallel portion and the number of applied processing units is made:

$$T = T_{sequential} + T_{parallel} \tag{2.1}$$

$$Speedup = \frac{T(1)}{T(n)} = \frac{T(1)}{T_{sequential} + \frac{T_{parallel}}{n}} = \frac{1}{P_{sequential} + \frac{P_{parallel}}{n}} \tag{2.2}$$

Therefore, he concludes that the portion of sequential work limits the performance of highly parallel processing. Figure 2.4a shows the expected speedup in runtime for an application with a parallel portion of 50%, 75%, 90%, and 95% that is executed on up to 4096 processing units.

Gustafson's Law In 1988, John Gustafson[46] expressed skepticism about Amdahl's Law and that the maximum speedup of an application is only $\frac{1}{P_{sequential}}$, even for an unlimited



(a) Scalability in runtime with fixed problem size depending on the portion of parallel work according to Amdahl's Law[3]

(b) Scalability in problem size with fixed runtime depending on the portion of parallel work according to Gustafson's Law[46]

Figure 2.4: Application speedup models

number of processing units. He also mentioned that in practice, one would not run a fixed-size problem on machines with a various number of processing units. Instead, the problem size generally expands in order to make use of the additional resources, which increases the parallel portion, because sequential work like initialization is only done once. Therefore, not the problem size, but the runtime is assumed to be constant when estimating the speedup. Figure 2.4b shows the scaled speedup in the problem size for a fixed runtime depending on the parallel portion of an application. The scaled speedup calculates as follows:

$$Speedup_{scaled} = \frac{P_{sequential} + P_{parallel} * n}{P_{sequential} + P_{parallel}} = n + (1 - n) * P_{sequential} \qquad (2.3)$$

Amdahl's Law and Gustafson's Law in the Multi- and Manycore Era Amdahl proposed to connect multiple single-processor systems to a distributed machine in order to cooperatively solve problems. Today, High Performance Computing (HPC) cluster machines are state of the art and even individual nodes contain numerous multi- or Manycore processors, making them a distributed system themselves. So, the scalability models can be applied on inter-node, intra-node, and on-chip levels. However, Amdahl and Gustafson did not take the overhead for, among other things, communication and resource management into account which also limits speedup[50]. Additionally, the software is not just infinitely parallel and sequential, but more complex in its structure. Superlinear speedup due to increased cache and main memory size when using more processing units is not considered as well. Other, more complex scalability models extend Amdahl's Law and Gustafson's Law by considering communication costs and available chip area as a limiting factor in multi-core systems[52]. Nonetheless, those models

only serve as a coarse approximation or an upper bound for the expected speedup, because this highly depends on the individual application and hardware. Often, parallel applications experience a sweet spot at a certain number of processing units and suffer from performance penalties when using more.

Speedup Efficiency Optimizing the number of allocated processing cores of a sub-linearly scaling application according to the shortest execution time maximizes the speedup, but reduces the overall efficiency of the system. The *efficiency* $E(n) = \frac{S(n)}{n}$ is defined as the normalized speedup divided by number of processors. Maximizing the efficiency and minimizing the execution time results in a conflict of opposing goals. The *speedup efficiency* $\eta(n) = E(n) * \frac{T(1)}{T(n)} = E(n) * S(n) = \frac{S(n)^2}{n}$ is a compromise for this conflict and describes the ratio of costs $C(n) = n * \frac{T(n)}{T(1)} = \frac{n}{S(n)}$ and benefits $E(n)$. The maximum speedup efficiency minimizes the cost-benefit ratio and is denoted as *processor working set*[40, 45]. Figure 2.5 illustrates the relationship between execution time, speedup, efficiency, and speedup efficiency.

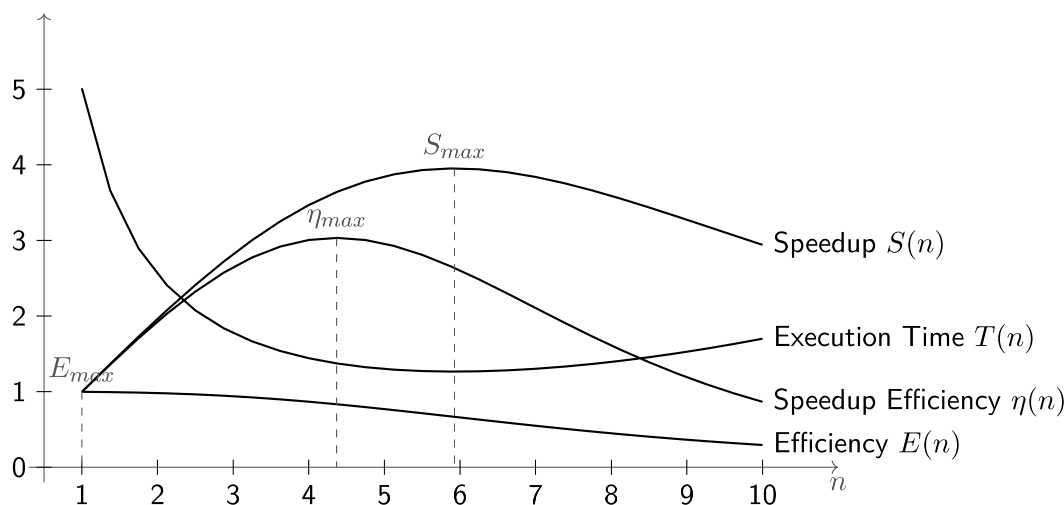


Figure 2.5: Relationship between execution time, speedup, efficiency, and speedup efficiency

2.3.2 Optimization Criteria

As already indicated, pure intra-process parallelism might not be sufficient to efficiently utilize the performance capabilities of future parallel processors. Hence, multiple processes can be executed simultaneously, sharing the same manycore processor. This requires suitable processor allocation management. Thereby, the primary goal of the processor allocation

is to ensure the successful execution of all applications, but there are further optimization goals. It is necessary to be specific about the goal in order to make appropriate allocation decisions[36].

Utilization The utilization is denoted as the percentage of processor cycles spent for productive computation or rather allocated to user programs. Optimizing only for utilization might lead to a long response time or even to starvation of small jobs, whereas massively parallel and long-running jobs will be preferred because they require a smaller timeshare for administrative overhead. This possibly harms the primary goal and is also limited to the current system load if below a certain saturation point. Additionally, utilization does not consider the efficiency of executing user code that might be able to occupy many resources but spends most processing time for management overhead.

Throughput The number of completed jobs per time unit is termed throughput. With an increasing number of jobs completed, more user applications become satisfied. Similarly to utilization, it is limited to the current system load. When saturated, a strictly throughput optimized system prefers short over long-running jobs which harms fairness and might cause starvation as well. Furthermore, the throughput heavily depends on the average size of the available jobs.

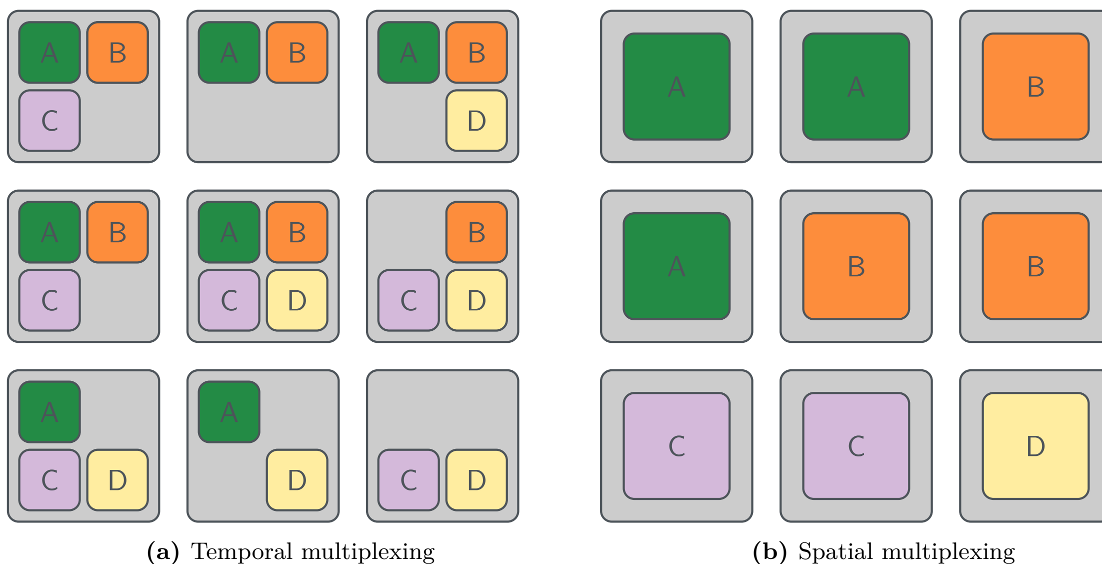
Response Time For interactive applications, fast feedback to the user is essential. Hence, two measures are relevant and thus serve as the basis for optimization. The *response time* denotes the latency from job submittal until first response to the user whereas the *turnaround time* is defined as the latency to completion. Analogically to throughput, the response time heavily depends on the job size. In addition, not all applications have time requirements and there are thresholds below which the user can no longer detect any improvement.

Fairness Resources are allocated fairly to all processes so that no process is permanently neglected. This reduces the average response time and avoids starvation. Fairness can be achieved on the user, process, or thread level.

Transparency Multiuser systems or systems on which the number of simultaneously running processes exceeds the hardware parallelism strive to create the illusion of a dedicated machine for each individual process. Being agnostic of other processes eases the software development and is usually achieved using temporal processor multiplexing which is examined in the following.

2.3.3 Processor Multiplexing

Modern parallel computer systems usually run multiple applications simultaneously to increase utilization. In order to enable this kind of resource sharing, mechanisms for partitioning the processing hardware are required. Thereby, a distinction between *temporal* and *spatial* processor multiplexing is made. The concepts of both types are depicted in figure 2.6.



(a) Temporal multiplexing **(b)** Spatial multiplexing

Figure 2.6: Types of processor multiplexing

Temporal Multiplexing Temporal multiplexing[32], which is also known as *time sharing*, denotes a processor sharing mechanism where multiple applications are allocated to a single processing element. Each application runs for some quantum of time before being preempted in order to allow the next application to run. This mechanism is required in multitasking OSs if the number of concurrently running applications exceeds the amount of available processing elements. Thus, a pseudo-concurrent execution can be achieved and the average response time can be minimized.

The procedure of changing the execution from one application to another is called *context switch* and needs to be performed after each quantum. The size of the quantum depends on the particular scheduling algorithm and the number of threads. Thereby, a too-large quantum time increases the response time which might not be tolerated in interactive environments, but a too small quantum size leads to unnecessarily frequent context switches, that introduce a certain overhead and thus reduce throughput [2].

Context Switch Overhead As mentioned before, context switches introduce overhead that reduces the proportion of processing time actually used for productive computation. This overhead arises from multiple sources. The direct overhead of a process context switch includes entering the kernel mode, switching the kernel stack, saving the old thread state, loading the new data into the registers, switching the address space, and refreshing the TLB. Additionally, indirect overhead arises due to missing hotness of the TLB and all levels of caches which slows down the execution of the new process after switching. While the direct overhead takes about tens of processor cycles, the indirect costs can be much higher but are heavily dependent on the hardware platform, application behavior, and data set size[72, 99]. Hence, frequent context switches introduce significant overhead to the actual productive work.

Modern processors contain Process-Context Identifiers (PCIDs) which are used to distinguish information about address translation cached for different address spaces. Thus, when creating entries in the TLB, it associates new entries with the current PCID and only uses matching entries for address translation. So, when switching contexts, the TLB does not have to be flushed and entries of processes might still be present from the previous execution[18]. However, the TLB owns a limited number of entries which are, if using temporal multiplexing, shared between all processes running on this processor and thus displace each other, reducing the hotness of the TLB.

Spatial Multiplexing Spatial multiplexing[32, 113], also called *space sharing*, refers to a processor sharing mechanism where the system is partitioned into clusters of processing elements, and each cluster is exclusively allocated to a single application that is allowed to run to completion without being preempted in favor of another application. Compared to temporal multiplexing, this reduces the overhead for context switching and eliminates side-channel effects caused by processor state sharing and thus increases the throughput and predictability of applications. Spatial multiplexing limits the number of simultaneously

running applications to the number of processing elements in the system, but future systems are expected to contain so many processing elements that they will exceed the average number of applications that a user wishes to run simultaneously[94]. Nevertheless, depending on the actual system load, parallel applications may get assigned fewer processing resources than requested. In order to avoid delays in execution, those applications must be able to dynamically adapt to the actual available resources. This property is also referred to as *resource-awareness* and requires applications to be *malleable* which is discussed in the following.

2.3.4 Spatial Processor Partitioning

As examined in section 2.3.1, the intra-program scalability is usually limited by multiple factors so that the speedup increases only sub-linearly with the number of processing elements which decreases the efficiency. So, in order to increase the system efficiency, inter-program scalability can be applied where the available processing resources need to be shared among multiple programs that are executed simultaneously. Spatial partitioning is a processor sharing mechanism where the available processing units are split into contiguous disjoint territories. Thereby, it focuses on the quantitative and qualitative allocation of processing resources to individual applications which are examined in the following[89].

Quantitative Partitioning

When partitioning a multi- or manycore system, one has to decide, which program obtains how many processors. The number of resources per application is determined by specific allocation strategies that strive to reach the optimization goal. The allocation decision can be made at different points in time.

Static Partitioning Using static partitioning, the number of processing units allocated to an application happens before the actual execution and stays constant over the entire runtime. The allocation can either be determined offline by solving user-defined constraints and compiler-generated information about the program structure and data dependencies or at the starting time of the application at which the current system load can be taken into account for decision making. Static partitioning has the advantage that applications can rely on the number of processor cores allocated to them during their entire runtime. Hence,

applications do not have to be able to release or accept additional cores at runtime. However, there are also disadvantages associated with static partitioning. If too many processor cores are currently occupied, additional applications can only be started after the execution of a running application has been completed and its cores become available again. Furthermore, running applications can not take benefit from the allocation of additional, otherwise unused cores. The resource allocation of running applications cannot be redistributed in order to optimize utilization or efficiency.

Dynamic Partitioning While in the case of static partitioning the programs were given a fixed number of processor cores over the whole runtime, dynamic partitioning is more flexible and allows additional cores to be allocated to and withdrawn from an application dynamically. So, the resource allocation can be optimized using resource redistribution in order to for instance, archive the highest speedup of individual applications or maximize global efficiency. Thereby, the overhead for profiling, reallocation, and adaption of the application needs to be lower than the expected performance gain. Dynamic partitioning requires co-design of the OS, the runtime system, and the application model. This is examined in the following section.

Parallel Application Types

Depending on the programming model, applications are expected to fulfill a resource handling model that meets the convention of the OS. Therefore, applications can be categorized into multiple job types[36] as shown in table 2.2.

Rigid Jobs Rigid jobs require a user-defined fixed number of logical processors that are statically requested at the creation time. The job is not able to run with less and cannot make use of additional processors, but it does not need to provide further information to the scheduler for decision making. Rigid jobs are typically applied for applications that are

Who decides	**When is it decided**	
	at submittal	during execution
User	*Rigid*	*Evolving*
System	*Moldable*	*Malleable*

Table 2.2: Parallel job types in terms of processor allocation[36]

written to be compatible with such simple system interfaces or are highly optimized for a fixed number of processors.

Moldable Jobs In contrast to rigid jobs, moldable jobs are flexible in the number of processors they require. They allow the scheduler to dictate the number of processors at the beginning of execution under the assumption that additional resources will improve the job's performance. The application automatically configures to the given number of processors and does not allow any reconfiguration of this number during execution. Hence, the scheduler is, based on the current system status, able to set the number of processors which is referred to as *adaptive partitioning*. This allows the system to globally maximize the overall system performance, while rigid jobs would only optimize to run at the sweet spot of their local speedup curve. Moldable jobs might specify individual constraints like a minimum and a maximum number of processors, memory, and response time requirements. A typical use case for moldable jobs is, for example Single Program Multiple Data (SPMD) style applications that can be executed over a wide range of processors.

Evolving Jobs Evolving jobs are similar to rigid jobs but may change their resource requirements during execution. Those are usually jobs that are composed of multiple phases with different parallelism properties. For each phase, evolving jobs can only continue execution when the current resource requirements are fulfilled and are not able to take benefit from additional resources.

Malleable Jobs Malleable jobs are the most flexible job type because they can adapt to changes in the number of processors during execution. This allows the scheduler to dynamically adjust the processor allocation when system load changes and is called *dynamic partitioning*. A varying number of assigned processor units can also be used to collect scalability information about the job during execution. Malleable jobs are currently only available in some scientific projects[68] but not supported on commercial machines because they require complex interactions between the OS and the runtime system. Most applications and OSs are neither able to express or handle dynamic resource limitations nor resource revocation.

Qualitative Partitioning

Besides the pure number of processing cores, the actual position in the network and other specific properties of the individual cores matter when allocating them to applications. Hence, qualitative partitioning concerns the question of which program obtains which processors[49].

Communication Distance In order to solve computational problems cooperatively, communication between processing units allocated to an application is required. Thereby, the performance depends on the communication delay, which increases with the physical distance and the number of routers that need to forward a message along the shortest path between the interacting processing units. Hence, communicating threads should be placed close together to keep communication delays low.

Fragmentation In order to minimize communication distance, each application should reside in exactly one individual, *contiguous* and pairwise disjoint territory, called partition. Allocating parts of a multi- or manycore processor as contiguous partitions leads to fragmentation. On the one hand, dynamic allocation and release operations may leave fractions of the many- or multicore processor unused because their size is too small to satisfy outstanding requests. This is referred to as *external fragmentation*. On the other hand, allocating partitions to applications that are larger than requested, in order to avoid external fragmentation and reduce contention on the NoC, causes *internal fragmentation* since a fraction of the allocated resources remains unused. Internal fragmentation can be handled by rearranging the partitions and consequently *migrating* the associated application threads which introduces significant overhead and disrupts the application processes. *Non-contiguous* allocation avoids fragmentation and thus allows to satisfy larger requests with a set of smaller partitions, which increases utilization but might cause long communication distances if not optimized according to the communication graphs of the applications. However, if the contiguous allocation fails, non-contiguous allocation can be applied complementary as a fallback mechanism.

Heterogeneity As mentioned in section 2.1, heterogeneous multi- and manycore processors integrate a mix of processing units that differ in power and performance characteristics or functionality. Accordingly, the suitability and performance according to the optimization goal for the particular applications need to be taken into account during resource allocation.

2.3.5 Application Profiling

When spatially partitioning multi- or manycore processors, it is necessary to decide how many cores to allocate to which application. This decision depends on the optimization goal and other factors such as the resource constraints of the individual applications. Thus, the following section investigates profiling approaches for parallel applications. This enables the comparison of parallel applications which is required for global resource allocation optimization.

Parallelism Profile

Applications usually run in multiple phases with varying degrees of parallelism. When executing a parallel application on a system with unlimited available processing units, then the histogram of the number n of active processor units over the execution time is denoted *parallelism profile*[89]. Figure 2.7 visualizes an example of a parallelism profile. The area under $n(t)$ indicates the amount of computational work and corresponds to the serial execution time $T(1) = \int_0^{T(\infty)} n(t)\,dt$. The average parallelism $\overline{n} = \frac{1}{T(\infty)} \int_0^{T(\infty)} n(t)\,dt = \frac{T(1)}{T(\infty)} = S(\infty)$ equals the asymptotic speedup when assuming an unlimited number of processor cores.

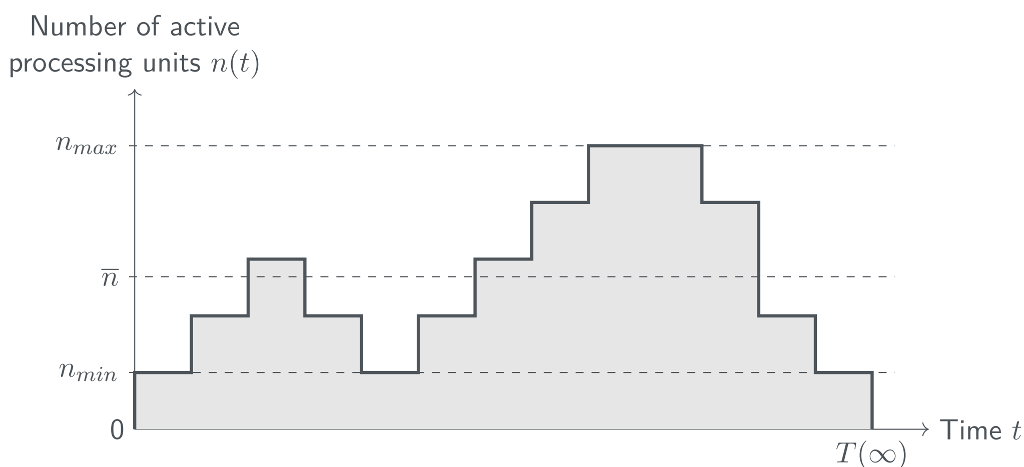


Figure 2.7: Parallelism profile of a parallel program

Speedup Model

Based on the parallelism profile, Downey[31] developed a scalability model that estimates the speedup curve of a parallel application using only the average parallelism $A = \overline{n}$ and σ which approximates the variance of parallelism. This unified scalability representation allows for comparison of the expected speedup gain depending on the assigned processing units for different applications and thus for allocation optimization when using inter-program parallelism. Equation 2.4 (taken from [31]) presents how the speedup is calculated. It differentiates a low variance and a high variance case as well as multiple cases depending on the number of allocated processing elements compared to the average parallelism. For a given application, the values A and σ can be determined by running them on the target platform.

$$\sigma < 1 \ \textbf{(low variance)}:$$

$$S(n) = \begin{cases} \dfrac{An}{A + \frac{\sigma}{2(n-1)}} & 1 \leq n \leq A \\ \dfrac{An}{\sigma(A - \frac{1}{2}) + n(1 - \frac{\sigma}{2})} & A \leq n \leq 2A - 1 \\ A & n \geq 2A - 1 \end{cases} \tag{2.4}$$

$$\sigma \geq 1 \ \textbf{(high variance)}:$$

$$S(n) = \begin{cases} \dfrac{nA(\sigma + 1)}{\sigma(n + A - 1) + A} & 1 \leq n \leq A + A\sigma - \sigma \\ A & n \geq A + A\sigma - \sigma \end{cases}$$

Offline Profiling

The application's parallelism profile and therefore the scalability parameters can be determined either statically, in advance of the actual execution, or dynamically during runtime. One approach for offline profiling is to execute the application to be profiled multiple times with a varying number of processing resources assigned on the target machine and measure the execution time. So, the speedup curve can be approximated. It is not necessary to measure the runtime for each individual number of processing units because missing values can be extrapolated or interpolated, but more measurement values enhance the accuracy[89, 27]. The preceding profiling in advance of the actual execution includes some obvious drawbacks.

Extensive profiling causes costs in the form of processor time and energy consumption which is difficult to be saved again during the optimized execution. Additionally, the parallelism profile can be highly dependent on the actual machine and the input data, which might not be statically known.

Online Profiling

In order to avoid the disadvantages of offline profiling, it can be replaced or combined with online profiling mechanisms. So, the preceding execution and the lack of knowledge about input data can be circumvented. The dynamic measurement of the parallelism profile is more sophisticated than the static analysis since the results are already needed for allocation optimization during runtime and profiling measurements are only useful after execution if the application will be executed again with the same configuration. In order to determine the speedup of an application dynamically while using the values for allocation optimization during this execution, direct or indirect approaches can be applied.

Direct approaches measure the application performance over specific intervals which are fractions of the application. Since the degree of parallelism in the phases of an application might change, the performance measurements over fixed time intervals are poorly comparable and are thus not accurate enough to determine the speedup curve. Alternatively, the performance over fixed periodic code intervals can be determined and compared. So, each time the program iterates over a specific loop procedure, the execution time is measured. This requires the periodicity of applications to be annotated which can be done either manually by the programmer or automatically by the compiler. The number of active processing units needs to be varied for several iterations to obtain a set of measurement points to approximate the speedup curve[39, 106].

In contrast to direct approaches, indirect profiling does not measure the speedup itself but derives it from the efficiency. As described in section 2.3.1, the efficiency is defined as $E(n) = \frac{S(n)}{n}$. An efficiency $E(n) = 1$ describes linear scalability and a speedup $S(n) = n$. The actual efficiency can indirectly be determined by measuring reasons that reduce the efficiency and subtracting that from the ideal efficiency. Causes of efficiency reduction are, e.g., resource sharing, cache coherency, synchronization, load imbalance, and parallelization overhead[34, 83, 84].

Modern processors provide hardware infrastructure, called Performance Monitoring Units (PMUs)[18], for counting performance-relevant events e.g. clock ticks, instructions, unhalted

cycles, and cache hits or misses. The availability and supported measurable event types are highly dependent on the particular processor architecture.

In this section, the theoretical basics of processor allocation have been examined. The following sections describe the practical implementation in widely used and scientific OSs.

2.3.6 Linux

Linux[104] is a Unix-based monolithic kernel that is the de facto standard OS on the world's most powerful computer systems[107]. It was considered a bottleneck on manycore systems but has shown good scalability on processors with tens of cores with minor adaptions[10, 73].

Thread Allocation Linux applies the abstraction of processes for running user programs and creates the illusion of many CPUs using processor virtualization which aids the program portability. Intra-program parallelism is supported using lightweight processes that share the same address space, namely threads. Threads are usually created through the C standard library that implements the Portable Operating System Interface (POSIX)[53, 90]. The procedure is as follows. The user program calls the *pthread_create* function which allocates and initializes memory for the stack and Thread-Local Storage (TLS). After that, the *clone* system call is invoked which creates a child process that shares parts of its execution context with the parent process. Nevertheless, a new Process Control Block (PCB) and kernel stack need to be allocated. The new thread is then handed over to the scheduler for execution[80, 104].

Completely Fair Scheduler The CPU scheduler of an OS is responsible to distribute the processor capacity among tasks. In Linux, the Completely Fair Scheduler (CFS) replaced the $O(1)$-scheduler and strives to provide fair processor allocation between tasks proportionally to their priorities and to maximize utilization while providing responsiveness for interactive applications using temporal multiplexing. Tasks are organized in a red-black tree that creates a timeline of future task execution in nanosecond granularity. The time-slices for each task are not constant but depend on the calculated fraction of the scheduling period, in which each task should run at least once to meet the target latency. The default scheduling period is set to 20 milliseconds. To prevent excessive scheduling when the number of tasks significantly

exceeds the number of processing units, CFS stretches the scheduling period so that the minimum time slice is four milliseconds[78, 95, 115].

CPUfreq Governors Linux implements infrastructure to operate dynamic voltage and frequency scaling mechanisms of the CPU, called *cpufreq*. This allows to save energy and scale CPU frequencies automatically, depending on the system load, in response to ACPI events, or manually by the user. Thereby, multiple power schemes (*governors*) are available to choose from. The *performance* and *powersave governor* statically set the CPU to the highest respectively lowest available frequency. The *ondemand* and *schedutil governor* scale the frequency dynamically according to the system load[88].

CPU Idle Governors While *CPUfreq* regulates processor power consumption during active phases (P-State), the *CPUidle* subsystem is responsible for managing the power of idling processors (C-State). As examined in section 2.2.3, sleep states enable significant energy savings during idle or partial load scenarios but choosing a proper idle sleep state requires a trade-off between potential energy savings and performance penalties for long respectively short phases of inactivity. The *CPUidle governor* implements a bundle of policies that control the processor using *CPUidle drivers*. For appropriate decision making, the Linux kernel implements a set of architecture-specific idle information, e.g. available C-States with their corresponding exit latency and target residency duration[55, 87, 104].

2.3.7 iRTSS

Invasive Computing[94, 103] investigates a paradigm for future parallel computing systems containing thousands of heterogeneous cores per chip. Therefore, hardware/software co-design is applied combining the development of a massively parallel Multi-Processor Systems-on-Chip (MPSoC) with resource-aware applications that explicitly express their preferred parallelism degree but are able to adapt to an actual available amount of resources. The underlying invasive hardware architecture is specified as tiled MPSoC where there are tiles with different functionalities like computation, I/O, and memory tiles. Heterogeneity can also be found within the individual tiles. Thus, compute tiles can contain a set of different compute elements like Reduced Instruction Set Computer (RISC) cores, dynamically reconfigurable invasive cores, or Tightly-Coupled Processor Arrays (TCPAs). All tiles are connected via an invasive NoC with a two-dimensional mesh topology. Cache coherence is only maintained

within the individual tiles. Invasive Run-Time Support System (iRTSS) denotes the system software that runs on top of the invasive hardware platform and consists of the *OctoPOS* parallel operating system and the *Agent System* that is responsible for dynamic resource management of the applications.

OctoPOS *OctoPOS*[85, 86, 103] is an OS for invasive computing and focuses on minimizing the overhead on massively parallel systems, enabling fine-grained parallelism. Aiming for high scalability and support of heterogeneous hardware, OctoPOS applies the design approach of a distributed OS and thus follows the example of *barrelfish*[7], *Popcorn*[5] and *M3*[4]. However, instead of replicating the OS for each core, it creates only a single instance per cache coherence domain and therefore per compute tile. In the context of Invasive Computing, the number of computing cores contained in future systems is expected to be greater than the average number of simultaneously running applications. Under this assumption, spatial processor multiplexing is applied as default following the one-thread-per-core execution model in order to reduce contention due to shared processor states of temporal multiplexing and improve throughput. Hence, the large number of cores are divided among the individual applications which get exclusive access to the allocated resources without any virtualization layer in between. Nevertheless, the limited number of computational resources requires the applications to behave resource-aware.

In order to exploit the high degree of parallelism from the hardware, OctoPOS provides the invasive application model, that is specially designed for tiled manycore systems, where each application requests and temporarily claims processing and memory resources, executes in parallel and frees resources afterwards. This enables fine-grained spatial expansion and contradiction of the resource-aware applications in the neighborhood of the actual computing environment according to their parallelism profile and reduces contention due to resource sharing. Figure 2.8 illustrates the life cycle of a resource-aware application running on OctoPOS. The central object of this procedure is called *Claim* and denotes a set of processing resources that are currently allocated to this particular application and can be used for parallel execution. Initially, a new claim is allocated by calling the *invade* system function. When the resource requirements of this application change, the *reinvade* function is used to adjust the current claim. The *assort* operation structures the program in parallel portions according to the claimed resources building a *Team* that represents an invasive-parallel program in execution and consists of a set of *i-lets*. Those are a lightweight control-flow abstraction and are typically executed using run-to-completion semantics sharing the same execution context, but are also able to block and cooperatively initiate a context switch to the

next i-let using lazy context allocation. This allows for fine-grain parallelism with minimal memory footprint. The team then infects the resources of the claim with the program and starts execution. After finishing execution, the application deallocates its claim and therefore the processing resource using the *retreat* function. So, the resources become available to other applications.

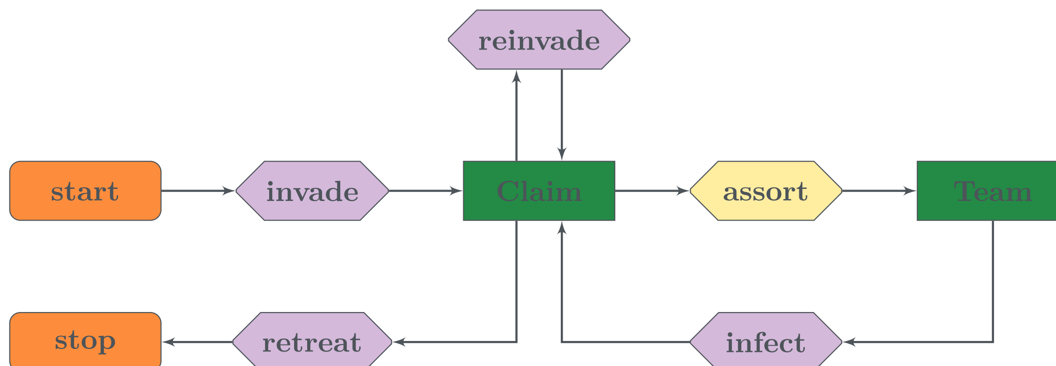


Figure 2.8: The life cycle of a resource-aware application in OctoPOS[85]

Agent System As a part of iRTSS, the *Agent System*[68, 67, 94, 103] runs on top of OctoPOS and is responsible for dynamic resource allocation to applications. It conquers the high complexity and scalability of scheduling highly dynamic and non-predictable workloads on manycore systems by using a distributed multi-agent-based resource management. Therefore, each application is advocated by an individual agent that continuously negotiates the resource allocation with its neighboring agents based on current hardware requirements and constraints. So, global throughput will be optimized while the agents make only local decisions, potentially providing shares of their resources to the neighboring applications that can reach the highest performance gains as long as the minimal resource constraints are fulfilled. Idle cores are managed by idle agents where initially one idle agent is created per 25 cores. An idle agent terminates when out of cores and terminating application agents, in turn, transform into idle agents again. The application profile is modeled using Downey's speedup model[31] (described in section 2.3.5) extended by considering NoC communication overhead and measured online using a *Core-i-let controller* in hardware per tile.

This distributed multi-agent approach corresponds to the concept of *self-stabilization*[28, 30] for *fault-tolerance* in distributed systems where a distributed system will end up in a correct state in a finite number of steps from any initial state. In this case, a correct state equals an optimal global resource allocation, which may get disrupted each time an application enters a new phase and thus changes its resource constraints.

2.3.8 MyThOS

This section provides an overview of existing abstractions and mechanisms of the *Many Threads Operating System (MyThOS)*[71, 81, 92] kernel that are necessary for the concept. MyThOS is a minimal OS and adopted many design concepts from *seL4* [66]. It follows the microkernel approach and promises to be a highly configurable and dynamically adaptable platform with a much lower thread creation latency compared to monolithic kernels like Linux. The communication to and between kernel objects is achieved through asynchronous messages that are executed in object-specific delegation monitors which ensure mutual exclusion and reduce cache misses due to local and synchronous execution of all messages in the queue of a specific object at the location of the current monitor owner. Table 2.3 lists corresponding concepts of other OSs compared to MyThOS kernel Objects. The most relevant kernel object types are examined in the following.

MyThOS	**seL4**	**Nova**	**Unix**
Address Space	Page Table	Protection Domain	Process
Capability Space	Cnodes		
Execution Context	Thread Control Block	Execution Context	Thread
Scheduling Context	?	Sched. Context	Scheduler
Portal	Endpoint	?	Sockets etc.
Frame	Frame	pages	mapped files
Kernel Memory	Untyped Memory	frame pool	frame pool

Table 2.3: Comparable concepts of MyThOS in sel4, Nova, and Unix[71]

Address Space The *Address Space* kernel objects represent a configuration for the translation from logical to physical addresses. It includes access protection flags and allows for mapping and unmapping physical memory frames into a logical address space.

Capability Space The application's access to kernel objects is managed using capabilities. A *capability space* implements a mapping from numerical capability pointers to capability entries that reference specific kernel objects and store meta-data such as access rights and resource inheritance information.

Execution Context *Execution contexts* are an abstraction for software threads of applications. They contain a copy of the thread state, including register contents as well as thread-local

storage, and offer mechanisms for suspension and migration. Each execution context is associated with an address space, a capability space, and a scheduling context that can be shared between multiple threads in arbitrary combinations.

Scheduling Context Individual logical processors, or more precisely hardware threads, are represented using *scheduling contexts*. An execution context can explicitly be bound to specific scheduling contexts to be scheduled at the specific place which corresponds to the thread affinity control known from common OSs. Thread migration is achieved by explicit rebinding of execution contexts to other scheduling contexts. The current implementation of scheduling contexts includes a cooperative First-In, First-Out (FIFO) scheduler that comes into operation when multiple software threads are bound to the same scheduling simultaneously. Hence, temporal multithreading with periodic preemption is not supported.

Portal *Portal* kernel objects act as endpoints for deferred synchronous communication between execution contexts as well as between applications and kernel objects. Each execution context requires its own portal including an individual message buffer frame in order to allow for concurrent capability invocations and Inter-Process Communication (IPC) requests. Each portal is associated with a specific execution context, which is resumed whenever a message arrives at the portal.

Frame *Frames* represent contiguous and well-aligned physical memory ranges that can be mapped into address spaces. Therefore, they can be used to establish shared memory across multiple address spaces. Frame objects can be inherited into sub-ranges and are responsible for tracking the use in address spaces. Additionally, they are able to enforce revocation by unmapping themselves from the affected address ranges.

Kernel Memory All kernel objects are explicitly allocated and do not change their size later on. This requires appropriate physical memory ranges but cannot be provided by frames because kernel objects are not allowed to be mapped to user-level address space. Hence, untyped *kernel memory* objects are used which, like frames, represent contiguous physical memory address ranges. Kernel memory can be split into smaller portions using resource inheritance.

2.4 Thread Management in Parallel Runtime Systems

Parallel applications strive to improve their performance by using multiple threads to solve problems cooperatively. Unfortunately, creating and starting a new thread is an expensive procedure when using standard OSs. Repeating this process every time a new fine-grain task is produced results in significant performance degradation, which contradicts the reason for using multiple threads. In order to mitigate this performance loss, common parallel runtime systems try to reduce the number of threads to be created by building and maintaining static thread pools instead of dynamically creating and destroying them. Accordingly, threads are created and kept in the thread pool until they are needed. After finishing a task, they are returned to the thread pool to be used again later[37, 22]. This behavior hides the dynamics in the application parallelism from the OS and, thus, complicates the resource and power management of the OS. To demonstrate this scheme, the thread management mechanisms of common parallel runtime systems are exemplarily examined in the following.

2.4.1 OpenMP

Open Multi-Processing (OpenMP) [37, 105] is a parallel programming Application Programming Interface (API) for C, C++, and Fortran on shared-memory systems supporting a wide range of OSs. If not defined otherwise in the environment variable *OMP_NUM_THREADS*, using the *num_threads* clause, or *omp_set_num_threads* library routine, then OpenMP automatically determines the number of logical processors and creates one worker thread per core. Idle workers register on a central thread pool and sleep in the OS kernel using *futex* mechanism via *pthread_cond_wait* on Linux or *WaitForSingleObject* on Windows until woken up again when new work arises or the process exits.

2.4.2 Intel Threading Building Blocks

Intel Threading Building Blocks (TBB)[22, 112, 91] is a task-based parallel programming library for `C++`. Instead of directly creating and programming threads, TBB offers multiple mechanisms to specify fine-grain tasks which are in dependable work packages that are scheduled using work-stealing[9]. If not manually configured differently, the TBB runtime system creates, in addition to the main application thread, one worker thread fewer than the number of logical cores available in the system. When a worker thread runs out of tasks to be executed, it follows a multi-level task dispatch loop trying to find an available task with

the highest affinity to this thread, before becoming idle. Idle worker threads are registered in a central thread pool and sleep in the OS kernel using a Fast Userspace Mutex (FUTEX) mechanism via *sem_wait* on Linux and *WaitForSingleObjectEx* on Windows until woken up recursively when new tasks become available or the process exits. Further implementation details are presented in section 4.3.

2.5 Conclusion

This thesis focuses on scalable and energy-efficient processor allocation mechanisms for multi- and manycore processor systems. Those processors strive to increase the overall peak performance and consume less energy than systems in which each processor has its own physical chip. CMOS is the standard technology for processor manufacturing and suffers from power dissipation that scales linearly with the switching frequency and the supply voltage squared. Due to the breakdown of Dennard Scaling, not all circuitry of those processors can be permanently active at the full frequency and the performance is therefore tightly coupled to the available energy budget. The energy consumption of CMOS processors can dynamically be adjusted using power gating, clock gating, and dynamic frequency and voltage scaling techniques which are partially controllable by the OS as available power states. The saved energy can be used to temporarily boost active cores. Consequently, such systems require the precise operation of the power adjustment infrastructure by theOS to avoid performance penalties and energy wastage.

In order to efficiently exploit the computational performance of multi- and manycore systems, multiprogramming has to be applied. Intra-program scalability is often limited and needs to be combined with inter-program parallelism. This requires temporal or spatial processor multiplexing. While temporal multiplexing requires regular preemption and thus context switches which introduce significant overhead and reduce the throughput, spatial multiplexing mechanisms necessitate co-design of the OS, the runtime system, and flexible applications when done dynamically. Therefore, quantitative and qualitative spatial partitioning ensures the successful execution of the applications and optimizes the overall system efficiency. To do so, the application's parallelism profile is needed and can be obtained using offline or online profiling approaches. However, the most existing OSs still rely on temporal multiplexing to form a convenient environment for applications on systems without massive parallelism. Dynamic spatial processor partitioning is only found in specialized OSs but requires applications to be profiled either offline or using specialized hardware. Additionally,

such systems lack preemptive mechanisms and rely on the cooperation of the applications to enforce redistribution decisions for global allocation optimization.

Creating new threads is an expensive procedure when using standard OSs. Hence, common parallel runtime systems maintain static thread pools, instead of dynamically creating new threads every time new tasks are generated and destroying them after execution. This behavior hides the dynamics in the application parallelism from the OS and, thus, complicates the resource and power management of the OS.

In conclusion, existing OSs are not suitable to handle the dynamics of parallel applications, which is the basis for efficient utilization of future manycore systems. Additionally, they lack proper dark silicon management that is required to maintain energy efficiency and avoid performance penalties. This will change in the following chapter.

CHAPTER 3

Energy-Efficient Processor Allocation on Multi- and Manycore Systems

This chapter explores approaches to conquer the lack of suitable processor allocation mechanisms for future multi- and manycore processors with a focus on scalability and energy efficiency.

Memory and processing units are both important resources to be managed by the OS and share multiple concerns. Hence, analogies between memory management and processor allocation are investigated in the following section. After that, the requirements, assumptions, and goals for the processor allocation concept to be developed are clarified. The concept itself is divided into three parts. At first, scalable and energy-efficient processor partitioning and allocation to applications are investigated. Second, online application profiling and resource redistribution mechanisms for global allocation optimization are explored. Spatial partitioning and resource reclamation require all applications to comply with the flexible execution model expected by the processor management in the OS. To disburden the programmer from manual resource management, a concept for integrating dynamic processor allocation into existing task-based parallel runtime systems is developed using work-stealing with dynamically sized worker pools. Finally, the main points of this concept are summarized.

3.1 Analogy to Memory Management

Processors and main memory are both important resources in computer systems that require accurate management in order to ensure successful execution and avoid performance losses. While the main memory size per node has already reached billions and trillions of bytes, the number of processor cores per compute node just started to steadily increase from tens to hundreds. In contrast to processor management, memory management[51] is, with respect to the number of resource units to manage, a well-studied field in computer science that has grown with increasing demand in size, bandwidth, and small latency of the main memory.

From a user's perspective, every programmer would like to have exclusive access to an unlimited number of infinitely fast processors that are tightly coupled over shared memory with Uniform Memory Access (UMA) properties. Due to physical constraints like limited energy density, increasing signal propagation delay with long distances in integrated circuits, manufacturing technology that defines the maximum number of transistors per die area, and production costs, the present technology does not provide such ideal systems. Instead, a rising number of energy-efficient processors that have interdependencies regarding package global power limitations are coupled via Non-Uniform Memory Access (NUMA) in order to ensure fast local memory access is common. Hence, managing those processors with respect to power limitations and data locality is an emerging field in operating system design. Therefore, this section explores design aspects of processor management that can be derived based on similarities to memory management.

3.1.1 Abstraction

In order to get any productive work done, hardware resources have to be exposed to user programs. Providing direct access to physical resources without any abstraction would force the user to care for resource management, which would, if not done right, lead to system crashes that cannot be prevented by the OS. Additionally, without resource abstraction, it is difficult to run multiple programs concurrently and thereby provide proper isolation between them to avoid influence.

For decades, *address spaces* have been used as an abstraction for main memory. Each program lives in an individual address space which allows for protection and relocation. The mapping of logical to physical memory addresses is realized by *segmentation* using *base/limit* registers and *virtual memory* using *paging*. Since programs are, due to isolation, agnostic of other programs, the total amount of memory needed can exceed the amount of physical memory available. This scenario is solved by temporally *swapping* parts or even whole address spaces onto disks, making the physical memory a time-multiplexed resource.

A commonly used abstraction for processors is *processes*. It was initially introduced to allow for pseudo-concurrent execution and to bridge over input and output latencies using multiprogramming in systems with only a single processor but is still applicable in systems with multiple CPUs[97].

3.1.2 Virtualization

In order to run programs that are not flexible in the number of resources available for execution, *virtualization* can be used, pretending to have more resources than physically existing. This concept is commonly implemented as *virtual memory* and used to extend the available main memory to the user at the cost of hardware and software management overhead and increased access latency if memory is swapped out to disk. Every program sees its private plain address space while the OS breaks it up into small chunks called *pages* and dynamically maps the logical to physical memory which is transparent to the user. Memory pages are used as a unit of allocation size and are typically from four kilobytes to one gigabyte large, which is a compromise between management overhead and fragmentation. When the usage is high, memory pages that need to be accessed in order to continue execution are loaded into the main memory, and pages that are not supposed to be used in the near future might get swapped out to mass storage. This is called *demand paging* and allows for functional memory extension but does not guarantee temporal transparency. This pays off due to programs usually running in different phases and with temporal and spatial data locality. Hence, only a part of a program's memory is really used and everything else can be swapped out during this time. Latencies introduced by loading data back from mass storage to main memory can be bridged using multiprogramming[26].

Using the same assumptions as for virtual memory, virtual processors can increase system utilization when pretending to have more processors than physically available. This approach equals the temporal multiplexing of processors in a multi-programmed environment where threads are scheduled alternately on processors. At this point, the goals for switching threads have to be differentiated. Rapid switching between threads helps to pretend pseudo-parallel execution of multiple sequential processes. This comes with periodic expenses in overhead for switching. In contrast to this, switching threads in the case of an I/O intensive program phase where the process is blocking, it is anyway an instrument to reactively bridge over processor dead times to increase utilization while only introducing switching costs when really needed.

Future manycore systems are assumed to have so many cores available that every program can occupy a set of private cores without the need for time-sharing. This makes it necessary to rethink virtualization with respect to management overhead for multiplexed resources. Removing processor virtualization from current systems requires support and a suitable interface from the OS and programs to behave flexibly in the number of resources available[94].

3.1.3 Replacement

At the latest when the system runs out of free resources, the operating system must decide which chunk of the logical resource will temporally be swapped out in order to fulfill the need of additional resources. The optimal replacement algorithm will always evict the resource that will be used the furthest in the future. This is not implementable since it requires the prediction of the future access pattern. In the context of memory management, this mechanism takes place as *page replacement algorithm* that is implemented in different flavors depending on the strategy. Examples for page replacement algorithms are First-In, First-Out (FIFO), Not Recently Used (NRU), Least Recently Used (LRU), Aging, Clock, and Working set algorithm[13, 35]. More sophisticated heuristics among these make their decisions based on the online profiling information of the application's memory access. Therefore, hardware features like, e.g., access information in the page table entries are used[44]. This corresponds to the dynamic profiling approaches used for processor allocation as examined in section 2.3.5. Replacement algorithms can be applied locally to the resources that are already allocated to a specific process or globally to the resources of all processes.

3.1.4 Working Set

For memory management, the working set[25] is referred to as a collection of currently referenced memory pages and serves as an approach for memory balancing in page replacement algorithms. The program can run with fewer memory pages but would suffer from severe performance degradation because of frequent page faults which each take the operating system up to a few milliseconds for handling.

The concept of a working set is applicable for processor allocation as well. In theory, all programs can be executed with only a single processor but they might suffer from performance loss that scales super linearly to the saved number of processors due to less available cache size and increased overhead of frequent context switches for communication and synchronization between tightly coupled software threads. As examined in section 2.3.1, the working set η_{max} in the context of processor allocation is defined as the number of processing units allocated to an application that maximizes the speedup-efficiency.

3.1.5 Fragmentation

An important issue that must be addressed with any allocator, including allocators for memory and processing resources, is the problem of fragmentation. External fragmentation is denoted as the inability to serve a user request for a contiguous block of resources because the available resources exist only in smaller and non-contiguous blocks. Internal fragmentation occurs when servicing a resource request with a block that is larger than the requested size since fractions of the allocated resources remain unused[44].

3.1.6 Resource Quality

When allocating resources, it is important to consider that the available resources do not always have the same properties and thus it has to be decided which resource to choose. For both memory and processor allocation, the relative positions of the processor core and the NUMA node to each other define the memory access latency which directly influences the performance. In addition, resource heterogeneity can not only be found in multi- and manycore processors, as described in section 2.1, but is for example also present in systems that contain both Dynamic Random-Access-Memory (DRAM) and Non-volatile Random-Access-Memory (NVRAM) as main memory.

3.2 Requirements and Assumptions

In order to design a suitable processor allocation mechanism, the design criteria must be defined. The goal of this thesis is to maximize the energy efficiency and thus the computation per watt ratio. For this, either the computing performance can be increased or the energy consumption can be reduced.

3.2.1 Hardware Model

As described in section 2.1, multi- and manycore processors come in different flavors. As already mentioned, heterogeneous processors integrate a mix of cores that differ in performance and power characteristics and functionality and can thus be used to further increase energy efficiency. However, this work concentrates on homogeneous processors, because others, e.g. [94], have already shown how resource management can be extended in order to support

heterogeneous hardware. Additionally, the memory architecture is expected to consist of one or multiple NUMA domains providing a global shared memory with cache coherency. Mechanisms for distributed memory machines and incoherent manycore systems are already addressed by others, e.g. [70, 94], as well. This thesis focuses on systems containing one or multiple sockets equipped with a multi- or manycore processor each. Any socket again can be structured in tiles containing a set of cores. Each core might provide multiple SMT threads. All cores are connected by a NoC.

3.2.2 Application Model

To exploit the potential of manycore systems, this work proposes multiprogramming in form of both inter-program and intra-program parallelism. All applications are expected to be *malleable* and thus need to be flexible in the number of available processing resources allocated by the OS. Applications start with a single hardware thread and can dynamically request further processing resources as required. Additional processing resources might be reclaimed by the OS due to resource redistribution.

3.2.3 Partitioning and Allocation Mechanism

Future manycore systems are expected to contain more processing elements than the average number of applications the user requires to run simultaneously on the system. Under this assumption, this work circumvents the overhead and the associated energy wastage for regular context switches of temporal processing multiplexing. Thus, the available processing resources are spatially partitioned and divided among the applications. Thereby, communication distances between processors of individual applications should be minimized.

To encourage applications and parallel runtime systems to dynamically allocate and release processing resources instead of maintaining static software thread pools, thread allocation latency needs to be minimized. This is also directly beneficial for performance and energy efficiency since the allocation might be in the critical path and hinders the application from making progress while waiting threads might waste energy. Since manycore systems contain more and more processor cores, more applications might run concurrently and also make individual allocation requests. Therefore, the scalability of the processor allocator is essential.

In order to further minimize the energy consumption and potentially accelerate active cores, processor cores that are currently not used should be put to sleep. Nevertheless, this must be united with a fast wakeup which is required to serve spontaneous thread allocation requests with low latency.

3.2.4 Profiling and Redistribution

The total processing units requested by all running applications might exceed the physically available amount and thus one has to decide which core to allocate to which application in order to maximize the energy efficiency of the execution. The same applies in the case of allocated but underutilized cores, which do not significantly increase the application's speedup but decrease the energy efficiency. Hence, the system's energy efficiency might benefit from a resource reclamation and subsequent redistribution or shutdown. Making proper decisions necessitates knowledge about the parallelism profiles of all applications.

This work does neither assume any static knowledge nor preceding offline profiling of the application's scalability. Hence, information is required to be gathered online during the actual execution. Additionally, information from the application, respectively the runtime system is not trusted, because it might be inaccurate and the information of different runtime systems might not be comparable. Consequently, application profiling needs to be done in the OS only using standard PMUs.

In order to enforce redistribution decisions, a resource reclamation mechanism is necessary. Besides the actual revocation of the processing unit, the associated application needs to be notified so that it has the opportunity to react and adapt to this decision.

The realization of the OS-level processor management depends on both mechanisms for the allocation and revocation of individual cores and strategies to decide which cores should when to be allocated to which application. In addition, the target sleep states for idling cores need to be determined. This work focuses on the mechanisms and provides interfaces to make strategies replaceable. The strategies that will be employed in this work for a proof of concept are not expected to be optimal.

3.2.5 Flexible Task-Parallel Runtime System

The application model requires user programs to be flexible in the number of allocated processor units and be able to handle revocation. This behavior can be implemented manually by the user, but increases code complexity and is potentially fault-prone. In order to relieve the user from the burden of manually managing dynamic resource allocation and revocation, this mechanism needs to be moved into the runtime system and thus be hidden from the user. So, productivity will be improved. This work focuses on task-based parallel runtime systems with work-stealing.

3.3 Hierarchical Processor Allocation

This section discusses the design aspects to develop a scalable and energy-efficient processor allocation mechanism for flexible, and therefore malleable, applications on manycore processors.

3.3.1 Processor Topology Tree

The power management of processors is tied to the actual hardware topology of the system. This also applies to the memory hierarchy and communication distances. Hence, this topology must also be modeled and made available to the OS. Figure 3.1 shows the tree-based structure of the processor topology model.

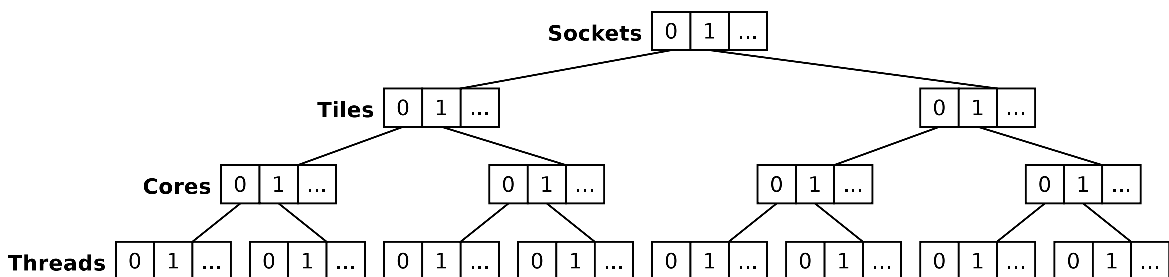


Figure 3.1: Structure of the processor topology tree

Since this work focuses on shared memory machines (e.g., a single compute node), sockets form the coarsest level of the processor topology. The resource management of an entire compute cluster might be implemented in the runtime environment on top of this local resource management layer. A compute node contains one or multiple processor sockets. As

already mentioned in section 2.1, manycore processors can be structured in tiles containing local caches, several cores, and form a management unit for dynamic voltage and frequency scaling. Each core might include multiple hardware threads. This model is applicable for all current multi-core and manycore systems. Due to its hierarchical structure, it eases the search for related processing elements which eases dynamic partitioning and promises scalability. Information about NoC topologies like two-dimensional meshes are not directly representable, but the communication distances between neighboring elements can be provided for each level of the topology tree.

3.3.2 Resource Management Approaches

Bookkeeping of unused processor cores, as well as the allocation to individual applications, can be managed in different places.

Central Widely used OSs like Linux usually employ centralized processor management mechanisms. Those provide a global view on the respective processor and task state which allows for global optimization of the schedule. While centralized approaches are comparably simple to implement, they suffer from limited scalability which contradicts the application on manycore systems[10].

Distributed Distributed resource management aims to avoid the scalability bottlenecks of centralized approaches. An example is multi-agent resource management (described in section 2.3.7) where each application is represented by an individual resource agent that continuously negotiates the processor allocation with neighboring agents. Idle cores are managed using idle agents[67]. Distributed management approaches offer the most possible scalability but lack global optimization due to the local view of the agents that is limited to the own neighborhood.

Hierarchical Hierarchical approaches create a trade-off between the global view of centralized and the scalability of distributed approaches. This promises not only sufficient scalability but also facilitates global energy consumption control because it is well compatible with the resource topology tree. Therefore, this work applies a two-level hierarchical approach for managing processor allocation. A central instance (*processor allocator*) keeps track of idling branches in the resource topology tree and allocates them to individual processes that are

represented as a *team* of threads when needed. Individual *thread teams* form the second level of the hierarchical management and maintain multiple lists for active and free resources on a smaller scale in chunk size and wakeup latency. The details of this mechanism are described in the following.

3.3.3 Hierarchical Processor Pools

Section 2.2.3 has stated that putting currently unused processor cores into an idle sleep state can significantly reduce energy consumption. Deeper sleep states promise the highest energy saving at the cost of increased wakeup latency. At first glance, this seems great for energy efficiency but contradicts the low allocation latency requirement. In contrast to this, keeping idling cores in an active state would lead to the lowest allocation latency but wastes much energy. Additionally, one has to consider that energy savings can be used for dynamic performance boosting of active cores, which also accelerates the application progress.

This work proposes a hierarchical set of processor pools holding processor cores in different sleep states to minimize both energy consumption and wakeup latency. Figure 3.2 illustrates the mechanism. The processor resources are provided as a machine-specific topology tree. In this example, we find a system containing two sockets, four tiles, eight cores, and 16 threads that are all evenly distributed over the system. The *processor allocator* is a central instance that manages all processing resources that are currently inactive and not allocated to a specific team (process). Therefore, it maintains a *processor pool*. Processor pools can reference individual threads or whole branches of the topology tree making it easier to partition the tree for power optimization and mapping strategy. The processor resources kept in the processor pool of the central processor allocator are not expected to be required in the near future and, therefore, are configured to enter a deep sleep state in order to minimize the power consumption.

The groups of processing resources that are assigned to individual processes are represented by a *team*. Each team contains three processor pools: *low latency free*, *cached*, and *active*. The low latency free pool holds hardware threads or branches of the topology tree that are currently not used and therefore in a low latency sleep state but already assigned to a specific team. The active processor pool of a team contains all threads that are currently executing code of the application. The cached pool holds processing resources that have already been active but are currently unused and entered a specific sleep state. When the application requests additional processing resources, the cached pool is the preferred source over the free

pool, because the cores have warm caches and, depending on the sleep state, cause a lower wakeup latency.

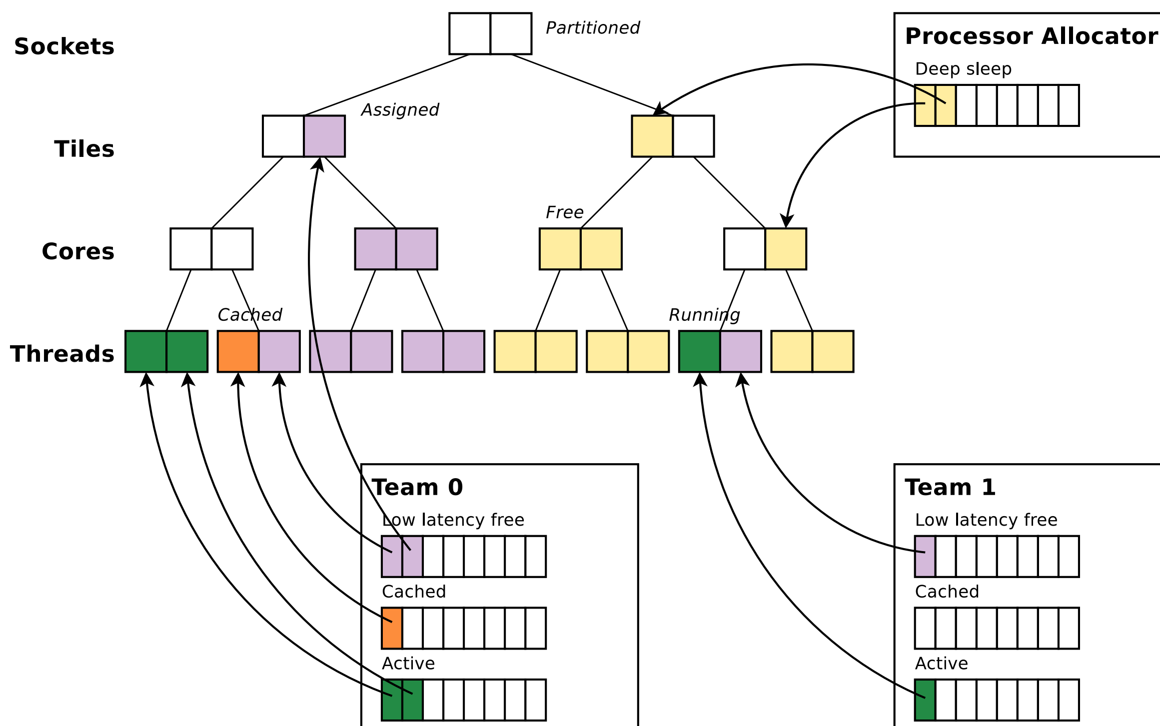


Figure 3.2: Processor pools

Figure 3.3 shows the state diagram of a processor thread under resource management. After initialization and when currently not needed, the processor allocator puts threads to deep sleep to reduce energy consumption. When needed, the pool balancing algorithm moves the ownership of a thread or a whole branch of the topology tree from the central processor allocator to a specific team that contains a local pool of free processing resources that are put into low latency sleep to be quickly available. After the application processor allocation request, a free thread is woken up, moved to the active list, and then starts executing code of the application. When a thread terminates its execution, it will be moved from the active list to the cached pool. When the whole application of a team terminates, all assigned processor resources are released to the central processor allocator. Additionally, if the pool balancing algorithm decides to do so, unused resources from the free and cached pools of a team can be revoked in favour of other teams or for energy optimization.

As already mentioned, each processor pool can reference arbitrary nodes in the topology tree. In order to provide a constant time lookup for the availability of a specific partition of the tree, each pool maintains an individual list for each of the four levels of the topology

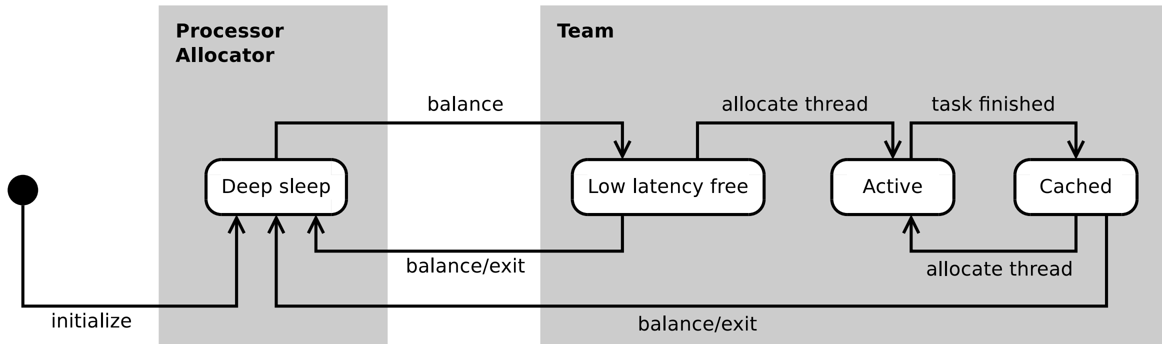


Figure 3.3: Resource management hardware thread state diagram

tree. The scheme is illustrated in figure 3.4. If, for example, a processor pool of the central processor allocator initially holds both of the two available sockets, they would be registered in the socket list of the pool. If then, a single thread is allocated from this pool, one of the socket entries will be split into its tiles, cores, and threads until the requested partition grain size is reached. Then again, if this single missing hardware thread is moved back to this pool, it will recursively merge with its associated threads, cores, and tiles. This is similar to the binary buddy memory allocation strategy. Hence, the pool can look up its biggest partition in constant time. In addition, it allows to simply move whole partitions as branches in a single step between different pools without moving each individual thread.

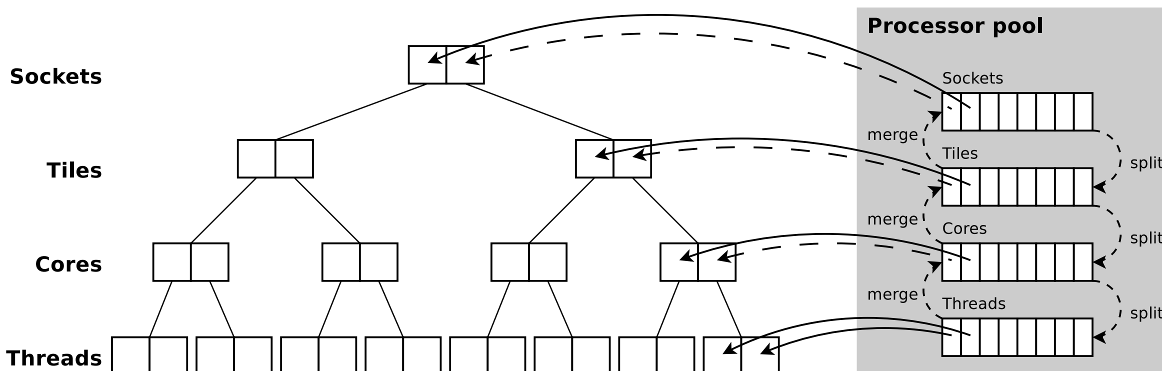


Figure 3.4: The internal structure of a processor pool

3.3.4 Pool Balancing Strategy

The proposed processor allocation mechanism operates multiple processor pools for different sleep states and applications. In order to decide which pool should own which resources at

what point of time and when should which resource be moved from one pool to another, a suitable pool balancing strategy is required.

When a logical processor terminates the execution of its software threads, it has to determine its target idle sleep state. Implementing the target sleep state to be pool-specific would require the processor to identify the pool in which it is currently managed. Each logical processor only knows its corresponding leaf in the topology tree and is able to derive the owner of the resource branch in which it is contained. Since each team includes multiple pools, determining the corresponding pool of a logical processor could either be done by searching all pools of the owner or continuously maintaining another reference for each logical processor in the topology tree to its assigned pool. In order to avoid this overhead, this work proposes to define owner-specific idle sleep states, because the ownership of a processing resource can directly be read from the topology tree. So, the central processor allocator as well as each thread team define the target idle sleep state for all processing resources they contain. This implies that all resources in both processor pools of a thread team have the same target idle sleep state.

Deep Sleep versus Low Latency Sleep The processor pool of the central processor allocator holds all processors that are currently not used and not assigned to any team. Therefore, they are put into a deep sleep state trying to save as much energy as possible. The processors in the pools of the individual teams are kept in a low latency sleep state to have a lower wakeup latency. In order to guarantee a fast response time of assigned threads when serving allocation requests, the low latency pools of the teams need to have a reserve of resources ready, whenever free resources are available in the system.

Global to Team For scalability reasons, each team maintains its own pool of free processors to serve frequent allocation requests without permanent interaction with the central processor allocator. Therefore, the central processor allocator generally provides bigger chunks of processing resources which are then managed at a fine grain level within the individual teams.

Team to Global When the system load increases, it can be necessary to reclaim unused processing resources from a team to be able to serve the resource requests of other teams. Consequently, the central processor allocator has to directly or indirectly inform the teams

about the system load which then triggers resource reclamation. On termination, all assigned resources of a team are deallocated back to the central processor allocator.

Strategy: Balancing Thresholds In order to avoid the overhead of frequent sampling, the pool states and speculative rebalancing which would require sophisticated heuristics, the applied rebalancing scheme only works reactively. Therefore, this work proposes to use a hysteresis for rebalancing and, hence, define thresholds for the reserve of resources in the pools. So, if the number of free processing resources in the pools of a team falls below this threshold, it triggers the rebalancing of the pools. Similarly, if the number of free resources in a team exceeds a specific threshold, some resources are returned to the central processor allocator. Those threshold values are platform-specific and considered as tuning parameters. Nevertheless, a shortage of free resources in the system or resource allocation adjustments due to global optimization can be realized using trimming of those thresholds for each individual team. Since this work focuses on the allocation mechanism, the developed pool balancing strategy is not expected to be optimal but is needed for a proof-of-concept.

3.3.5 Placement Strategy

As described in section 2.3.4, besides the pure number of processing units, the actual position in the network and specific properties of the cores matter as well. So, when allocating processor resources, the question of which resource to assign to which application in which order arises. This qualitative partitioning is realized through *mapping* or *placement* decisions and has a major impact on the application performance and energy consumption[74]. Thereby, effects like communication distances, network contention, memory hierarchy, energy consumption, heat distribution, and fragmentation have to be considered. Since this work focuses on homogeneous multi- and manycore systems, placement optimizations due to heterogeneous compute units are not examined.

Communication Distance and Network Contention Parallel applications usually run multiple software threads that need to communicate with each other in order to solve problems cooperatively. Communication distance in the NoC and therefore latency between interacting threads heavily influences the performance. Hence, tightly coupled threads need to be placed close together in terms of network distance to ensure fast interaction. On the other hand, placing too many threads in the same area of the network might overload the

communication channels and slow all threads down due to network contention. Consequently, preferably independent clusters of threads need to be placed at a sufficient distance to other threads.

Memory Hierarchy The same characteristics, as for communication distance and contention, also apply to the memory hierarchy. Threads of the same application are assumed to work on shared data and therefore profit from shared caches with low distance and therefore low access latency. If, however, the currently used data of a group of software threads in the same area of the system exceeds the capacity of the caches, they become a performance bottleneck. In this case, distributing this group of threads over a wider area in the system increases the usable cache size and thus lowers the pressure on the cache but potentially increases the access latency for data that resides in remote caches of other hardware threads.

Energy Consumption and Heat Distribution Putting unused processors into a sleep state reduces energy consumption. Since the hardware power management operates on the granularity of the individual branches of the processor topology tree, it seems reasonable to place all software threads close together in the branch in order to keep as many branches of the topology tree as possible in a deep sleep state. Nevertheless, placing all software threads close together would reduce the performance of the active cores due to the lack of heat distribution in the system, which intensifies the dark silicon effect and accelerates the hardware wear out.

Fragmentation The allocation mechanism serves the spontaneous resource requests of the individual applications. From a user's perspective, the chunk size of a requested processing resource from the processor allocation mechanism in the OS is always a single logical hardware thread. This prevents external fragmentation of the processor allocation when assuming the same properties for all processing resources. However, due to implicit resource sharing on multiple levels, processing resources may not be assigned independently. So, in order to avoid unintentional influences and contention between unrelated applications, each application should reside in a contiguous pairwise disjoint partition of the manycore processor system. Due to possible spontaneous and unpredictable allocation requests from all applications during execution, contiguous allocation causes fragmentation. If the contiguous allocation fails due to external fragmentation, non-contiguous allocation can be applied complementary as a fallback mechanism to further serve requests. This results in sub-optimal placement due to increased

communication distance and access latency to data in remote caches. Internal fragmentation and non-contiguous allocation can be handled using defragmentation. Thus, one has to decide whether to keep a sub-optimal allocation or resolve it subsequently by migrating the software threads of different applications within the system. Making the decision for thread migration depends on the expected performance improvements and migration overhead that includes the interruption of running threads and replacement of cached data. In addition, due to migration latency and possibly changing properties of the execution location (e.g. unsynchronized clocks), thread migration is not transparent to the user and therefore might hinder the progress of all threads assigned to its application. Since the processor allocation demand can possibly change anytime, it needs to be estimated by heuristics to predict future requirement changes in order to determine the costs and benefits of thread migration.

Maximum Serial Performance Placement Strategy From the perspective of a single application, one has to decide whether to place threads close together or distribute them widely over the partition. Placing all software threads close together would lead to a bad utilization of the cache capacity, increased network contention, and reduced performance of the active cores due to the lack of heat distribution in the system, which intensifies the dark silicon effect. So, distributing the software threads widely increases the serial performance, because it avoids resource contention. Additionally, this might be more energy-efficient since high serial performance might cause an application to finish its execution earlier. So, all processors can again enter a deep sleep state or be used to process the following applications. However, in order to avoid contention with other applications, a wide distribution requires a large partition size and causes internal fragmentation, because close processing elements remain idle in favor of the serial performance.

Maximum Resource Sharing Placement Strategy In order to maximize the systems utilization, the software threads of individual applications can be placed closely. So, communication distances within the NoC and access latency to shared caches are minimized. This is beneficial for tightly coupled threads whose performance is primarily determined by communication overhead. Additionally, internal fragmentation in the partitions can be reduced so that a greater fraction of the system remains available to other applications or can be put to sleep to decrease energy consumption. A dense arrangement of software threads maximizes resource sharing and thus the system's utilization. Nevertheless, it is vulnerable to contention which is able to heavily degrade the performance of individual threads.

Intra-Application Locality Placement Strategy This work does not presume knowledge about applications' communication schemes or memory access pattern, but expect different applications not to communicate regularly with each other. Thus, individual applications should reside as far apart as possible to reduce resource contention and the risk for external fragmentation of future contiguous allocation. Software threads of the same application, on the other hand, are suspected to tightly interact with each other and are therefore placed closely. This minimizes communication distance, maximizes resource sharing, and reduces internal fragmentation within the partitions.

This work focuses on the processor allocation mechanism and therefore does not have the claim of an optimal placement strategy. As a proof-of-concept, the described application-local placement strategy is developed and implemented but is designed to be exchangeable. The algorithm can be described as a combination of the memory management strategies *binary buddy* for splitting the topology tree, *first fit* for the allocation of application threads within a partition, and *worst fit* for placing a partition for a newly created application.

3.4 Application Profiling and Processor Redistribution

As examined in section 2.3.5, when using dynamic partitioning, the question of how many processing resources to allocate to which application arises. When leaving this decision to the application developer, they will strive to execute at the knee of their speedup curves and thus optimize the allocation only according to their local point of view. This approach breaks at latest if the sum of required processing elements of all simultaneously running applications exceeds the hardware capabilities and applications need to compete for resources. In addition, it is prone to inaccurate speedup estimations or malicious resource requests of applications. Thus, this thesis proposes a system global allocation optimization to maximize the overall system performance and energy efficiency. Consequently, individual applications can be executed at a point in their speedup curve that might differ from their local optimum in favor of global efficiency. However, making proper allocation decisions necessitates knowledge about the parallelism profile of all applications. In order to be independent of possibly inaccurate and hardly comparable information from the application and to disburden the user from static analyses, profiling needs to be done online by the OS. So, online profiling techniques are investigated in the following. Afterward, it is examined how redistribution decisions can be enforced using resource reclamation.

3.4.1 Online Application Profiling

The profiling approach described in this section is based on the master thesis of Florian Bartz[6] that was supervised during the creation of this work. That thesis investigated online profiling mechanisms for dynamic processor partitioning. Therefore, multiple concepts for determining the current application acceleration by scaling the assigned number of processors were discussed. It proposed an approach that is based on an indirect determination of the application's parallel efficiency by means of performance counters and can serve as a basis for decision-making in dynamic partitioning. The evaluation has shown that passive waiting and its impact on parallel efficiency can reliably be detected but cache coherency related delays were only determined inaccurately.

Online Profiling Approaches

Besides offline profiling, static preferences of the application programmer, or statistics of the parallel runtime system, there are multiple approaches to determine applications' parallelism profile, without having to rely on user information. These include repetition of program sections while varying the number of processors , directly measuring the application progress, and indirect efficiency determination which are described in the following.

Varying the Number of Processors for Program Sections As described in section 2.3.5, offline profiling mechanisms are able to determine the scalability by executing a given application multiple times with a varying number of processing elements assigned. This approach is applicable for online profiling within the OS as well. Instead of measuring the execution time of the whole application, the speedup of specific program sections is determined by varying the number of allocated processing resources per iteration. So, the speedup curve can be approximated. Although this approach seems simple to implement, it has some disadvantages. It requires user annotation or detection of sections and thus the program structure. Since different sections in the program vary in the amount of work, only measurements of the same sections can be compared. Therefore, the application is required to have an iterative structure. Additionally, the size of the inspected sections has to be small enough to allow allocation adjustments before the application finishes execution but also has to be big enough to allow accurate measurements and avoid the high management overhead for profiling of too many fine-grained sections. Due to changing input data, synchronization, or I/O latency, scalability results of such program sections can be distorted.

Measuring the Application Progress The execution time of an application depends on how long it takes to execute its instructions. Therefore, one might try to measure the execution speed of an application in terms of instructions per time interval. So, the speedup could be determined by varying the number of allocated processing units and monitoring its influence on the instruction execution rate. However, the pure number of executed instructions is not a reliable indication for the progress of an application execution, because it is not fixed and not every instruction contributes to the application progress. A large number of executed instructions can also be caused by a high management or parallelization overhead of the parallel runtime system rather than by a high execution speed of the application. Even with active waiting, instructions are executed without causing any progress in application execution. Nevertheless, some certain instructions, e.g. floating point operations, reflect the progress more reliably, because they are usually used only rarely by the parallel runtime system. Still, the occurrence of those instructions is highly application-specific and cannot be used as a general indicator.

Indirect Efficiency Determination The direct determination of an application's speedup suffers from the difficulty of measuring the progress accurately and the need for comparative values. Therefore, this work follows the approach of deriving the speedup from the efficiency, which can indirectly be determined by measuring influences that reduce the efficiency and subtracting them from the ideal efficiency. Thereby, the ideal efficiency assumes a linear scalability of the application. The advantage of this approach is that it does neither require comparative values for speedup, nor multiple executions of specific code sections, nor any cooperation by the application. However, the accuracy depends on the determination of influences that degrade efficiency. Those are examined in the following.

Reasons for Inefficiency

The indirect determination of speedup, that is derived from the efficiency, requires accurate measurement of the influences that degrade efficiency. This can be done without assistance and transparent to applications by the OS using hardware monitoring infrastructure. This work assumes only standard PMUs for profiling. Unfortunately, the availability and supported measurable event types are highly dependent on the particular processor architecture. Nevertheless, a survey of efficiency influences in parallel applications is given in the following.

Waiting Due to synchronization, I/O latency, or if workers run out of tasks, associated software threads need to wait until further processing is possible. Waiting wastes processor cycles which thus contradicts the utilization and efficiency of the processor. In the case of passive waiting, the affected processor unit halts its execution which is directly measurable by the PMUs. During active waiting, on the other hand, the software thread continuously checks the fulfillment of the blocking condition. This causes permanent utilization of the processing unit without generating application progress and is difficult to detect, because the executed instructions barely differ from the instructions potentially used in other sections of an application.

Parallelization Overhead Parallelization overhead results from the effort required to initialize and coordinate the parallel execution of applications and is considered to increase with the number of used processing units. From the hardware perspective, the instructions that are executed for this parallelization are indistinguishable from other instructions of the application. Therefore, the parallelization overhead is not determined in this work.

Resource Sharing When executing applications in parallel, shared hardware resources can affect efficiency through positive and negative interference. In the case of positive interference, the execution time can be reduced because an operation is performed by a shared resource only once for several program sections which would have to be executed multiple times if the application was executed sequentially. One example for this are main memory requests whose data is accessed by multiple threads and collaboratively used through the shared caches. Positive interference allows for super-linearly scalability and thus for an efficiency greater than one. However, the execution time savings are difficult to determine because it requires to measure how much more often operations on shared resources would have to be executed in the case of sequential execution and how much execution time is required for those operations. The effects of positive interference are therefore not determined in this work. On the other hand, there are negative interferences that reduce efficiency and thus cause sub-linear scalability. In this case, additional latency is introduced by parallel running threads that compete for shared resources. An example of negative interference is data in a shared cache that needs to be reloaded from main memory because it has been displaced from the cache due to memory accesses by other processor cores. With negative interference, additional time is required because several application sections running in parallel compete for a shared resource. Some of those waiting times can be determined at runtime using PMUs

but the precise scope of measurable interference due to shared resources depends on the specific processor.

Dynamic Partitioning Strategy

As discussed previously, this work derives the speedup of an application from its efficiency which is indirectly determined by measuring causes of inefficiency. However, such application profiling information needs to be translated into allocation and redistribution decisions to increase the system's productivity and thus energy efficiency. Hence, a suitable dynamic partitioning strategy needs to be employed. As mentioned in section 3.2, this work focuses on mechanisms rather than strategies. Thus, the applied partitioning strategy is not expected to deliver optimal results but is used as a proof of concept and considered exchangeable. An overview about state-of-the-art dynamic partitioning strategies can be found in the master thesis of Florian Bartz[6].

Equipartition (EQUI)[75, 108] is a space sharing strategy that strives to maintain equal allocation of processors to all applications. This provides fairness but does not take the individual characteristics of the applications parallelsim profiles into account and thus results in suboptimal system efficiency. The *self-tuning equipartition* (ST-EQUI) strategy[84] uses runtime measured speedup characteristics to dynamically adjust the partitions from the applications local point of view. Therefore, each application regularly estimates how many of its processors it should actually use to maximize its speedup. If an application benefits from fewer processors than currently allocated, it releases the unused processors to the system, which then reallocates them equally among the other applications. In contrast to this, the *equal efficiency* (EQUAL-EFF) strategy[84] strives to maximize the global system efficiency instead of running applications at the knee of their local speedup curves. Therefore, the system allocates most processors to those application that achieve best efficiency, but still this does not necessarily mean a good efficiency. To conquer this issue, an improved version of the equal efficiency strategy (EQUAL-EFF++)[15] only allocates additional processors if an application satisfies a target efficiency. Consequently, some processors may remain unallocated leading to a reduced utilization of the machine. This can be avoided by dynamically adjusting the number of running applications to the system load[16].

This work applies the EQUAL-EFF++ partitioning strategy because it promises high system efficiency which is expected to also optimize energy efficiency. Underutilization due to unallocated processors will not directly be targeted, but unused processing units are put into

an energy saving idle state to reduce power consumption and boost active cores to further increase energy efficiency. However, adjusting the number of allocated processors requires resource revocation from inefficient applications. Hence, resource revocation mechanisms are discussed in the following.

3.4.2 Processor Revocation

When, due to application profiling and ongoing optimization, the processor management decides on redistribution of the resource allocation, then a mechanism for revocation of assigned processing resources is required. Simply withdrawing an assigned processor and thus stopping the execution of the corresponding software thread potentially harms the functional integrity of an application, since this event is hardly detectable by the application. Additionally, this might cause a deadlock because other software threads will at some point wait for a response of the suspended thread. Therefore, an application needs to be informed in case of resource revocation, giving it the opportunity to react and thus restore functional integrity. Still, the procedure of notification and revocation can be done either cooperatively or preemptively.

Cooperative Revocation From an application's perspective, it is a convenient approach to become just notified about the intention of the withdrawal of a processor and then have the possibility to bring the corresponding software thread in a safe state and to terminate or migrate it afterwards by yourself. This facilitates the synchronization within an application and thus reduces the risk of deadlocks. The downside of this cooperative revocation scheme is that, from the resource management point of view, the response time until a revocation is realized heavily depends on the behavior of the individual application and is thus not statically known. Additionally, faulty or malicious applications might not behave cooperatively and consequently do not return such resources. Hence, resource redistribution decisions cannot be enforced and the processing units will not become available for other undersupplied applications.

Preemptive Revocation To guarantee immediate enforcement of redistribution decisions and to avoid the dependence of cooperative behavior of applications, the resource revocation can be realized preemptively. Therefore, the allocated processing unit is directly revoked from the application and the corresponding software thread becomes suspended accordingly.

Only afterwards the application will be informed and can then handle the revocation by, e.g., migrating the suspended software thread to another processing unit and safely terminating it there. Nevertheless, migrating suspended threads to just exit them directly afterwards introduces additional overhead compared to cooperative termination. However, the actual revocation handling routine needs to be application specific.

Although cooperative processor revocation simplifies the handling for applications and reduces the deadlock risk, this work applies a preemptive revocation scheme, because it guarantees instant enforcement of redistribution decisions and is robust against malicious application behavior.

3.5 Dynamic Processor Allocation in Task-based Parallel Runtime Systems

The previously described processing resource allocation and redistribution mechanism requires applications to dynamically allocate processing resources and handle revocations to allow for energy-efficient allocation optimization. To disburden the application programmer from the manual resource handling, this behavior can be implemented in the runtime system. Exemplarily, this section explores how existing task-based runtime systems with work-stealing can be adopted to meet those requirements. Therefore, the following section first investigates the dynamic processor allocation followed by the dynamic worker suspension and resumption.

The worker suspension mechanism described in this section is based on the master thesis of Oliver Giersch[41] that was supervised during the creation of this work. That thesis investigated work-stealing with dynamically sized worker pools and thus mechanisms to suspend workers in case of processor revocation.

3.5.1 Dynamic Thread Allocation

In order to take full advantage of the potential of multi- or manycore processors, applications need to split their work and employ multiple threads in parallel to process a problem cooperatively. Parallel runtime systems ease the development of parallel applications, because they implement the basic functionality of the execution model. This includes, e.g., taskification, synchronization, work distribution, and work balancing. Hence, the runtime system is

responsible to distribute the application's tasks over the available processor units. This can be done using multiple approaches that bring different implications for dynamic resource management.

Static Worker Pool

As examined in section 2.4, existing task-based parallel runtime systems usually maintain static worker pools to avoid the significant overhead of regular processor allocations on common OSs. Unfortunately, this behavior hides the application's dynamics from the processor resource management. In addition, when using the proposed processor allocation mechanism without temporal multiplexing, idling workers would block their corresponding processors from being reassigned to other undersupplied applications. This results in underutilization and thus reduced energy efficiency. The proposed profiling and redistribution mechanism strives to detect this reduced efficiency of the application in phases of low utilization and will penalize it using processor revocation. Nevertheless, this profiling and redistribution procedure is only enforced with a certain delay so that valuable processing time and energy is wasted by idling workers. Then again, if the application enters a program phase with high parallelism, it may be denied further resources because of inadequate profiling statistics. In conclusion, the static worker pools are not suitable to achieve maximum energy efficiency in conjunction with the proposed processor management.

One Thread per Task

Static worker pools are employed to avoid the overhead of regular thread allocations. The proposed processor allocation mechanism strives for low thread allocation latency which reduces this overhead. Hence, frequently allocating threads becomes less costly, which again raises the question whether it is reasonable to just allocate an individual thread per task. This would make work distribution strategies in the runtime system unnecessary as long as more processing units than simultaneous tasks are available. Additionally, the dynamics in the parallelism of the application would directly be observable by the processor management. However, if application programmers employ the fine-grain task parallelism that is supported by existing runtime systems, the number of simultaneous tasks might exceed the number of available processing units. Thus, work distribution strategies are still needed. Furthermore, although the costs for thread allocation are reduced they still accumulate with the number of created task. In conclusion, dynamically allocating one software thread and

thus one processing unit per task created provides maximum interaction with the processor management but introduces significant overhead if the task size is small. Nonetheless, too fine-grain parallelism information are not helpful for the processor management because its time scale does not allow redistribution nor significant energy saving through dark silicon management between tasks.

Adaptive Work Stealing

There are approaches which dynamically adjust the number of workers in task-based parallel runtime systems according to the actually available amount of work and other factors as , e.g., observed efficiency in the runtime system or feedback from external resource management components. Those are commonly termed *adaptive work stealing*[1, 29, 110]. Therefore, new workers are dynamically added to the work stealing domain by allocating the necessary resources if the available work load increases while underutilized workers are removed. With adaptive work stealing, the high overhead of separate thread allocation per task can be reduced while still providing the application's dynamics to the processor management. In addition, this avoids the potentially low utilization and lack of dynamics of the static worker pools and allows to handle resource limitations from the processor management. The downside of adaptive work stealing is that it trades dynamicity for increased complexity because it requires the collection of feedback data upon which it has to decide how to adjust the worker allocation. A survey about state-of-the-art adaptive work stealing algorithms can be found in the master thesis of Oliver Giersch[41].

This work proposes to employ adaptive work stealing to dynamically negotiate the resource allocation with the processor management based on the actually available work and the system load. It promises to be a reasonable approach to disburden the application programmer from manual processing resource allocation while still providing information about the application's dynamics in parallelism to the processor management.

3.5.2 Worker Suspension

As described in section 3.4.2, assigned processing units need to be revoked if the proposed resource management decides to. This processor revocation can be realized using cooperative or preemptive approaches. In order to disburden the application programmer from manual reaction of resource revocation, the question of how to handle those events automatically in

the parallel runtime system arises. When using a work stealing based task scheduler in the runtime system, there is one worker thread assumed to be running per allocated processing unit. Hence, the worker thread of a revoked processing unit needs to be suspended without corrupting the functional integrity of the application. There are multiple approaches for suspending worker threads which are discussed in the following.

Cooperative Suspension

A straightforward approach for worker suspension is to simply indicate a suspension request to the worker and wait for it to cooperatively terminate. So, the worker is able to finish its current task. Remaining tasks in the queue of the worker can either be individually stolen by other workers or the whole queue can be set muggable and be assigned to another idling worker[1]. In case of cooperative suspension, all workers need to regularly check the revocation indicator at certain points of their scheduling loops. If a suspension request is detected, the worker makes its remaining tasks available to other workers and halts its execution in a safe state for later resumption or terminates. Cooperative suspension can be used to dynamically adjust the worker pool size due to adaptive work stealing, but is only suitable for cooperative revocation mechanisms. It is not applicable for preemptive revocation schemes, because the response time until a worker becomes suspended depends on the currently running task size and is thus not statically known.

Preemptive Suspension

In contrast to cooperative suspension, preemptive suspension enforces an instant suspension of a worker and does not depend on the size of the currently running task. Therefore, when a suspension is requested, the worker is halted immediately and its execution state, including the register set and stack, is saved. Pending tasks are available for execution by other workers. Since the suspended worker was probably interrupted during the execution of a task, the task remains unfinished and needs to be continued later. This requires worker resumption mechanisms and therefore probably a migration of the suspended worker to another processing unit to continue and finish the interrupted task. Preemptive suspension is a suitable mechanism to automatically handle preemptive processor revocation in parallel runtime systems with work stealing.

Hybrid Suspension

While cooperative worker suspension suffers from a potentially long response time in case of long-running tasks, preemptive suspension requires later worker resumption and thus introduces migration overhead which might exceed the remaining execution of a small task. To ensure a static response time of a suspension request and avoid migration overhead of small tasks, a hybrid suspension scheme can be used. Therefore, a cooperative suspension is requested firstly. Only if the worker does not become suspended after a certain time, a preemptive suspension is triggered. Although the hybrid suspension combines the benefits of cooperative and preemptive suspension, revoked processing units do not become instantly free and sophisticated time management for outstanding revocation requests is required.

This work proposes to apply a plain preemptive suspension, because revoked processing resources are instantly made available for other purposes. Additionally, it is not prone to faulty or malicious applications.

3.5.3 Worker Resumption

Preemptive suspension instantly halts the execution of the worker thread, which requires later resumption to continue and finish its current task. Figure 3.5 illustrates the procedure of worker resumption. The state of a suspended worker is defined by its corresponding thread context that includes saved stack and register contents. The associated task queue of the worker might be empty because the remaining tasks could be stolen by other workers during suspension. To continue and finish the task that the worker was executing at the point of interruption, the worker needs to be rescheduled to a processing unit. This can be initiated either by the system scheduler or by another worker thread.

Resumption by Scheduler

One approach for worker resumption is that a worker which was originally suspended by the system scheduler is also resumed by the scheduler. Therefore, the worker thread is moved to a ready list of the system scheduler. When a free processing unit becomes available, the worker is rescheduled and continues execution. This can either be the case if other applications release processing resources or will occur at the latest if another worker thread of the same application runs out of tasks and terminates due to underutilization. Hence, a processing

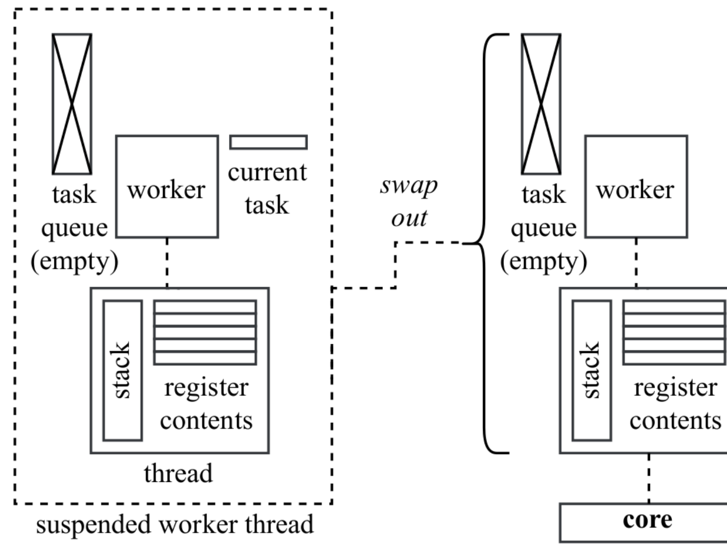


Figure 3.5: Procedure of resuming a suspended worker thread by rebinding its saved software thread context to a processing unit, taken from [41]

unit becomes available and the suspended worker can be resumed without requiring any intervention by the runtime system. If the worker is assigned to a processing unit in same area of the system where it was previously executed, the data locality can be maintained.

Resumption by Worker

When relying only on worker resumption by the scheduler, idle workers of the same application will usually perform multiple rounds of work stealing attempts before terminating, wasting valuable processor cycles until the processing unit is released and handed over to the suspended worker. To avoid this, idle workers could directly resume a suspended worker, if available, instead of trying to steal tasks from active workers. Therefore, suspended workers need to be declared muggable in the runtime system to be resumed by idling workers.

This work proposes a combination of resumption by a worker and by the scheduler. Hence, the suspended workers are registered to the system scheduler demanding to be assigned to a free processing unit. When, in the meantime, another worker becomes idle, then it hands over its processing unit to the suspended worker. In this way, a suspended worker will directly be resumed in both cases if an additional processing resource becomes assigned to the application or if another worker thread of the same application becomes idle.

3.6 Summary

Existing OSs are not suitable to handle the dynamics of parallel applications, which is the basis for efficient utilization of future manycore systems. Additionally, they lack proper dark silicon management that is required to maintain energy efficiency and avoid performance penalties. To conquer this deficiency, processor allocation mechanisms for future multi- and manycore processors with a focus on scalability and energy efficiency were investigated in this chapter. Thereby, analogies to memory management were made, because memory and processing units are both important resources that require accurate management to ensure successful execution and avoid performance degradation.

Although multi- and manycore processors come in different flavours, this work focuses on systems containing one or multiple sockets equipped with a homogeneous multi- or manycore processor each. Any socket again can be structured in tiles containing a set of cores which are conected by a NoC and may provide multiple SMT threads. Additionally, the memory architecture is expected to consist of one or multiple NUMA domains providing a global shared memory with cache coherency.

Future manycore systems are expected to contain more processing elements than the average number of applications the user requires to run simultaneously on the system. Under this assumption, this work circumvents the overhead and the associated energy wastage for regular context switches of temporal processing multiplexing. Thus, the available processing resources are spatially partitioned and divided among the applications. This requires applications to be malleable and thus flexible in the number of available processing resources allocated by the OS.

The power management of processors is tied to the actual hardware topology of the system. This also applies to the memory hierarchy and communication distances. Therefore, the processing resources are modeled and managed using a tree-based topology structure. Scalability in the processor allocation mechanism is reached by using a two-level hierarchical management approach. Therefore, a central processor allocator keeps track of idling branches in the resource topology tree and allocates them to individual processes, that are represented as a team of threads, when needed. Individual thread teams form the second level of hierarchical management and maintain multiple processor pools for active and free resources on a smaller scale in chunk size and wakeup latency.

To encourage applications and parallel runtime systems to dynamically allocate and release processing resources instead of maintaining static software thread pools, the developed allocation mechanism strives to minimize the thread allocation latency. This is also directly beneficial for performance and energy efficiency since the allocation might be in the critical path of execution and hinders the application from making progress while waiting threads might waste energy. To further minimize the energy consumption and potentially accelerate active cores, processor cores that are currently not used are put to sleep. Using multiple processor pools with different idle sleep states, energy efficiency and low latency thread allocation can be united.

When using dynamic partitioning, the question of how many processing resources to allocate to which application arises. When leaving this decision to the application developer, they will strive to execute at the knee of their speedup curves and thus optimize the allocation only according to their local point of view. Thus, this thesis proposes a system global allocation optimization to maximize the overall system performance and energy efficiency. Allocation decisions are made using the EQUAL-EFF++ partitioning strategy[15] based on online application profiling using indirect efficiency determination. This work applies a preemptive revocation scheme, because it guarantees instant enforcement of redistribution decisions and is robust against malicious application behavior.

The application model requires user programs to be flexible in the number of allocated processor units and be able to handle resource revocation. This behavior can be implemented manually by the user, but increases code complexity and is potentially fault-prone. In order to relieve the user from the burden of manually managing dynamic resource allocation and revocation, this mechanism needs to be moved into the runtime system and can thus be hidden from the user. Hence, productivity will be improved. Therefore, this chapter investigated how the required execution model can exemplarily be integrated into task-based parallel runtime systems with work stealing. It has been discussed that adaptive work stealing can be applied to dynamically negotiate the resource allocation with the processor management based on the actually available work and the system load. This work proposes to apply a plain preemptive worker suspension in case of resource revocation, because revoked processing resources are instantly made available for other purposes and it is not prone to faulty or malicious applications. Suspended workers are later continued using a resumption mechanism to finish their current task.

The following chapter provides an overview of how the developed mechanisms can be implemented and integrated into existing OSs and task-based parallel runtime systems.

CHAPTER 4

Implementation

This chapter provides insight into the implementation details of the developed processor allocation and redistribution mechanisms and is structured as follows. Dynamic processor partitioning requires co-design of the OS and applications to allow for on-demand resource allocation. Hence, section 4.1 examines how the widely used Posix-thread API can be extended to enable proper resource limitation while reducing necessary modifications of the application software and changes in the habits of programmers. After that, an overview of the software architecture of the allocation mechanism and its integration into MyThOS is provided. Optimization of processor allocation to maximize energy efficiency requires dynamic resource redistribution based on application profiling. Therefore, section 4.2 examines how application information are gathered and describes the realization of the redistribution mechanism. Finally, section 4.3 demonstrates the exemplary integration of adaptive work stealing and resource revocation handling into TBB to meet the execution model while disburden the application programmer from manual resource management.

4.1 Processor Allocator

This section provides a brief overview about the implementation of the previously developed hierarchical processor allocation mechanism. In order to reach a high scalability and low overhead, the proposed processor allocation mechanism is integrated in the minimal and highly parallel Many Threads Operating System (MyThOS)[81, 92]. As examined in section 2.3.8, it follows the microkernel approach and promises to be a highly configurable and dynamically adaptable platform with a much lower thread creation latency compared to monolithic kernels like Linux. The communication to and between kernel objects is achieved through asynchronous messages that are executed in object specific delegation monitors which ensure mutual exclusion and reduce cache misses due to local and synchronous execution of all messages in the queue of a specific object at the location of the current monitor owner. Kernel object access protection is achieved using capabilities. However, MyThOS itself lacks

a processor allocator and thus leaves the management of the resources entirely to the user. So, it forms a suitable basement that can benefit from the extension of the developed processor management approach. Before presenting the integration of the proposed processor allocator in MyThOS, the interface to the user is examined.

4.1.1 User Interface

Due to the absence of processor time-slicing, the processor allocation mechanism requires the user applications to act resource aware. Hence, they must be able to handle rejected resource allocation requests. In addition, in case of insufficient resources, it is beneficially for an application to express a demand of additional resources to the OS, so that further resources can automatically be assigned when available.

Listing 4.1: Extended posix thread interface

```
#include <pthread.h>

// behavior desired by the user in case of insufficient resources
typedef enum {
  // fail and return 0
  CREATE_FAIL = 0,
  // run thread next to other thread using time-sharing
  CREATE_FORCE = 1,
  // enqueue to wait list and run when free resources become available
  CREATE_DEMAND = 2
} pthread_core_alloc_t;

int pthread_create(pthread_t *thread,
      const pthread_attr_t *attr,
      void *(*start_routine)(void *),
      void *arg,
      pthread_core_alloc_t allocType = FAIL);

int pthread_revoke_demand(pthread_t *thread);
```

To reduce necessary modifications of application software and changes in the habits of programmers, the processor allocation mechanism is made accessible through an extended Posix-thread interface that is integrated into the *musl libc*[80](Version 1.1.20) environment. Listing 4.1 shows the extended posix thread interface. The function *pthread_create* accepts an

additional parameter (*allocType*) that indicates how the processor management, as part of the operating system, should behave in case of insufficient resources. For compatibility reasons, the default value of this parameter is set to *CREATE_FAIL* and thus *pthread_create* fails with error code *EAGAIN*, if there are currently no free resources available. Therefore, this behavior is equal to the original Posix API. Unfortunately, some common parallel runtime systems like OpenMP[37] and TBB[22] do not adapt their resource requirements when resource allocation using *pthread_create* fails, but abort execution and terminate the application. Calling this function with *CREATE_DEMAND* instructs the resource management, in case of insufficient resources, to store the pthread in a waiting queue and automatically schedule it to a new free processor when available. When, in turn, the resource requirements change again and the demanded pthreads are no longer needed, they can be removed from the waiting queue using *pthread_revoke_demand* function. In order to handle race conditions, this function returns a specific value to indicate whether a thread has successfully been removed from the waiting queue or has been scheduled meanwhile. Using *pthread_create* with the argument *CREATE_FORCE* causes resource management to schedule the pthread next to another pthread on the same logical processor using time-sharing. This eases the adaptation of application software that cannot dynamically adapt the number of required threads. Thereby, to avoid the overhead of frequent context switches, time-sharing is implemented using a cooperative scheduling scheme rather than fixed time-slicing.

4.1.2 Architecture Overview

Now that we have shown the user interface, we explore cooperation with the processor allocation mechanisms developed. Therefore, figure 4.1 provides a simplified overview of the architecture and communication dependencies of the involved software components. As mentioned previously, the application expresses its requirements of processing resources through an extended Posix API that is implemented in a *musl libc* library. *Musl*[80] is a *C standard library* and is intended to be used for operating systems that are based on the Linux kernel. Therefore, it communicates directly with OS using the Linux system call interface in order to access basic functionality of the OS like memory or thread management. This contradicts the microkernel approach of MyThOS where only a minimum amount of software is executed in kernel mode and some traditional OS functions are typically moved to user space. Additionally, MyThOS internally uses a different abstraction for software threads than Linux which hinders direct interactions. To overcome this issue, a *compatibility layer* is introduced as a mediator between the *musl libc* and MyThOS. It intercepts and

emulates necessary system calls such as *mmap*, *clone*, and *exit*. Furthermore, when required, it communicates with the objects that need to be implemented in kernel mode. While communication of user-space modules is still synchronous, communication between user-space and kernel, as well as inside the kernel, is asynchronous to achieve high scalability.

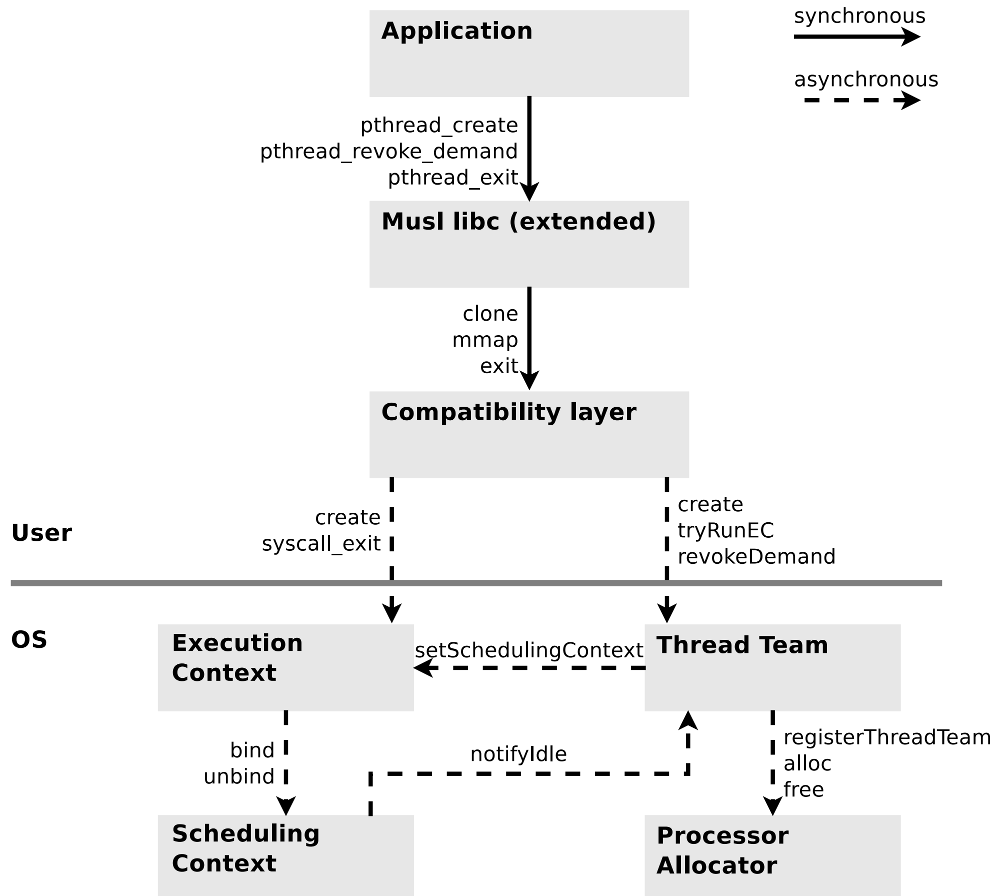


Figure 4.1: Processor allocation architecture overview

The proposed processor resource management is implemented inside the OS kernel, because it requires fast interaction of multiple kernel objects. Hence, a single processor allocator kernel object maintains the topology tree where each leaf references its associative scheduling context. When a new user processes is created, a thread team kernel object is spawned accordingly in order to represent and manage the group of processing resources related to this specific process. This is required due to the fact that process creation is implemented as a user level library function and the OS is thus agnostic to those structures, but needs knowledge about related threads to make its resource management decisions. Whenever a process allocates a new software thread, an execution context needs to be created and

configured. After that, the user requests its thread team to allocate a free scheduling context and bind the execution context to it. If no free processing resources are available in the thread team, it requests additional resources from the central processor allocator. On termination, the execution context unbinds from the scheduling context which then notifies its assigned thread team of being idle.

4.1.3 Thread Context Allocation

In order to process a user application, every allocated processor requires a context. A context includes user memory for stack and TLS, as well as kernel memory to save the thread state in case of interruption, and, depending on the OS implementation, an individual kernel stack. As described in section 2.3, context creation and usage are handled differently in the individual OSs.

In Linux[80, 104], for each software thread, a user context with stack and TLS and a kernel context with its own stack are allocated one after the other. The individual kernel stack allows blocking in kernel mode during interruptions or system calls so that the processor can switch to other processes. Hence, dead times can be bridged using temporal multiplexing. The disadvantages of these heavy-weight contexts are the huge memory footprint and the high overhead for memory allocation each time a new thread is spawned.

MyThOS[71, 81, 92] does not support preemptive but only cooperative multi-threading, and threads are not allowed to block in kernel mode because system calls are implemented in a run-to-completion semantic. Thus, software threads do not need individual kernel stacks. Hence, there is only a single kernel stack per logical processor owned by the associated scheduling context. This decreases the amount of memory needed per software thread. Despite this, a user stack, TLS, and thread state including register contents must be allocated per software thread.

OctoPOS[85, 86, 103] uses an light-weight control-flow abstraction called *i-lets*. Those have a minimal memory footprint and are typically executed using run-to-completion semantics sharing the same execution context. However, *i-lets* are allowed to block and cooperatively initiate a context switch to the next *i-let* using lazy context allocation. So in the best case, each application needs to create only one execution context per processing unit it owns.

This work follows the one-thread-per-core execution model as default but supports cooperative multi-threading in exceptional cases. Thus, a single kernel stack per logical processor is

sufficient. Nevertheless, execution contexts for individual software threads are allocated on demand, because they are application-specific due to address space isolation and need to be configured anyway.

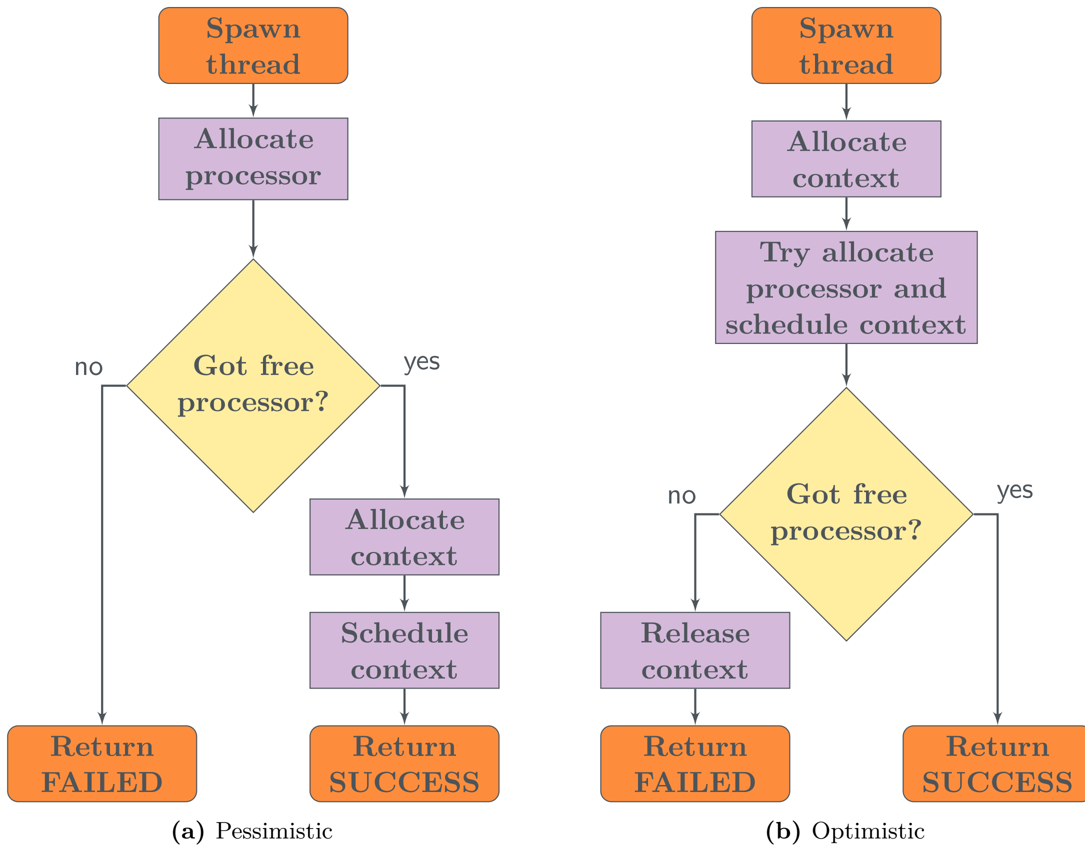
One goal of this work is the minimization of the allocation latency to encourage the user to use dynamic thread allocation. With respect to limited resource availability due to spatial partitioning, the question of what to allocate first, processor or context, arises. The order depends on whether one is optimistic or pessimistic about the availability of a free processor. Figure 4.2 illustrates the corresponding thread allocation procedures.

Pessimistic Context Allocation

A pessimistic thread creation procedure behaves skeptically toward the successful allocation of a free processor. Consequently, it tries to minimize overhead in case of insufficient processing resources. So, it allocates the context only if a processor is available. If, however, a free processor can be obtained, a context needs to be allocated anyway and can be scheduled on this processor afterward. This requires the processor to be reserved and thus idling until a context for execution is provided. If, in turn, the context allocation fails because, e.g., the application runs out of memory, the allocated processor needs to be released. Due to this structure, the processor allocation and scheduling of the context need to be done in separate steps, which also results in individual system calls. Additionally, the reservation of processors without context to be scheduled is prone to user faults because the processor remains reserved and consequently blocked if the user misses the deallocation after a fault.

Optimistic Context Allocation

If being optimistic about the availability of free processors, the allocation latency can be minimized for this case. So, the context is allocated first and the processor allocation and scheduling of the context can be done in a single step. This reduces the number of interactions, and thus system calls, with the resource management of the OS. In addition, intermediate reservation of the processor is not necessary. If, on the other hand, the processor allocation fails, the just allocated context needs to be released again, which increases the wasted execution time for an unsuccessful thread allocation. Nevertheless, this work applies optimistic context allocation, because processor allocation and context scheduling can be combined into a single step.



(a) Pessimistic **(b)** Optimistic

Figure 4.2: Comparison of pessimistic and optimistic context and processor allocation procedures

Context Recycling

Optimistic thread creation requires a context allocation regardless of the actual availability of free processing resources. Hence, a low latency context allocation is essential. The latency of frequent context allocation and deallocation in highly dynamic applications can be reduced by recycling contexts. This trades the allocation latency for memory needed to retain released contexts. So, if available, an existing context can be reconfigured, instead of creating a new one. Since contexts contain memory for the user stack and TLS, which are mapped into the address space of a specific application, contexts can only be reused for new threads of the same application without costly remapping of this memory. This results in application-wise context pools. Contexts can be organized with or without relation to the processor on which they were last executed. On the one hand, limiting the recycling of contexts to the previously used processor can be beneficial for performance since the caches of this processor might still contain parts of the context. On the other hand, this potentially wastes more memory for contexts if the processor allocation changes and it is not applicable in the case

of optimistic context allocation, because the actual processing element is not known at the context allocation time. Therefore, this work applies context recycling using application-wise context pools without processor binding.

4.2 Application Profiling and Resource Redistribution

Section 3.4.1 proposed to apply the EQUAL-EFF++[15] dynamic partitioning strategy to limit the partition size of each individual application based on the measured efficiency. It promises high system efficiency, which is expected to also optimize energy efficiency. However, this strategy requires parallel applications to first achieve a target efficiency in order to receive more processors, which will introduce a certain delay in execution time until a sufficient amount of resources becomes available to highly parallel and efficient applications. In addition, this does not take phases with different parallelism and efficiency of an application into account. Therefore, this work uses a different implementation. The partition size of an application will only be limited when the measured efficiency falls below the target efficiency. To adjust the maximum partition size accordingly to the continuously measured efficiency and to prevent oscillations due to the resulting feedback loop, a Proportional–Integral–Derivative controller (PID controller)[8, 47] is used. The control loop is illustrated in figure 4.3.

User applications dynamically allocate and release processing resources according to their individual parallelism. The amount of currently assigned cores is represented as the processor partition. A periodic timer with a 100 milliseconds interval regularly triggers the efficiency measurement of all processors that are currently included in the partition. This interrupts all cores, which then asynchronously gather efficiency statistics using their local performance monitoring facilities. The average efficiency of the partition is determined and then compared to the target efficiency. The difference between these values is passed as the input error value to the PID controller. In order to reduce the error, this controller calculates a correction value using proportional, integral, and derivative terms. The correction value is used to limit the maximum partition size of the individual application, if the target efficiency is not achieved. Regardless of the measured efficiency, each application owns at least one processing unit to ensure successful execution. The behavior of the PID controller is configured by the coefficients for the proportional, integral, and derivative terms: k_p, k_i, and k_d. The actual configuration values used in this work are described in section 5.6.1.

If the amount of processing resources assigned to an application exceeds the resource limitation from the PID controller, surplus cores are revoked from the application. Therefore, the cores

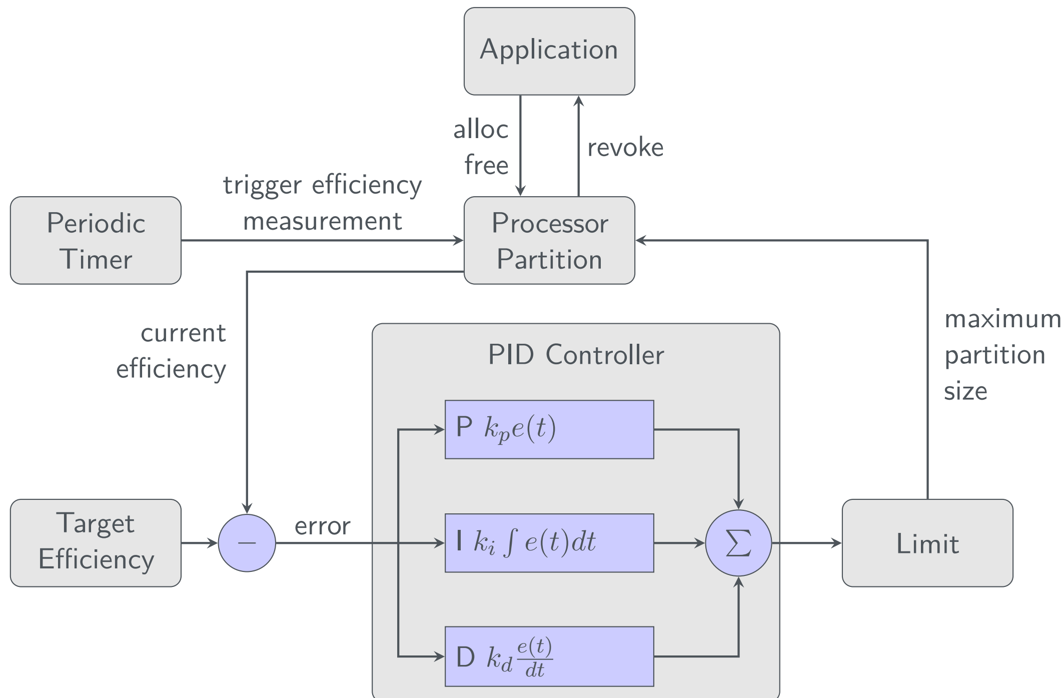


Figure 4.3: Control loop to limit the partition size of an application based on the measured efficiency

are interrupted, and the user execution contexts are unbound from corresponding scheduling contexts, that represent the cores. So, the execution of user code is halted on those cores and the thread states are saved. As examined in section 3.4.2, this work proposes to immediately inform applications about resource revocation, giving them the opportunity to react and thus restore functional integrity. This is implemented using a signaling mechanism that enqueues revocation events into an event buffer and activates a specialized execution context for signal handling. This execution context is then scheduled with high priority at the scheduling context of the main application thread to ensure instant event processing. The signal handler executes an application-specific revocation handler.

4.3 Dynamically Sized Worker Pools in Threading Building Blocks

As examined in section 3.5, this work proposes to employ adaptive work stealing in the runtime system, to dynamically negotiate the resource allocation with the processor management based on the actually available work and the system load. It promises to be a reasonable approach to disburden the application developer from manual processing resource allocation while still providing information about the application dynamics in parallelism to the processor

management. Therefore, Intel Threading Building Blocks (TBB)[22, 91, 112], as an example for a task parallel runtime system, is modified to allocate workers, including processing units, on demand, and terminate idle workers so that the corresponding processing units are released to the resource management again.

4.3.1 Dynamic Worker Allocation

The TBB library[22] is structured in a client layer, that is responsible for task creation and execution, and a resource management layer, that provides worker threads to clients according to their current amount of parallel work. By default, the resource management layer of TBB creates, in addition to the main application thread, one worker thread less than the number of logical cores available in the system at the beginning of the first parallel section. This logic is mainly implemented in the *tbb::internal::rml::private_server* class. The implementation of the underlying software threads is kept platform independent, but *tbb::internal::rml::thread_monitor* uses Posix threads for Linux-based systems which are also employed as an interface for thread allocation in this work. In case of unsuccessful Posix thread creation, the whole program is terminated. When a worker thread runs out of tasks to execute, it follows a multilevel task dispatch loop that tries to find an available task with the highest affinity for this thread before becoming idle. Idle worker threads are registered in a central thread list and sleep in the OS kernel using a FUTEX via *sem_wait* on Linux until recursively woken up when new tasks become available or the process exits.

The dynamic partitioning approach of this work requires applications to handle limited processing resources. Immediately terminating the process, as it is TBB's reaction to low availability of resources, is a suitable way to deal with this issue if this case almost never occurs because common OSs with time sharing do not limit the number of software threads to the actually available amount of hardware processing units. However, the developed resource management mechanism increases the probability of unsuccessful Posix thread creation due to limited resources. In this case, finishing the program with only the available processing units is preferable. Therefore, TBB is modified to put uninitialized worker threads back to the idle list and continue execution without them in case of unsuccessful Posix thread creation due to limited resources. Putting idle worker threads to sleep until needed again avoids the overhead of regular thread allocations, but hides the application dynamics from the processor resource management, complicates dark-silicon management, and prohibits the reallocation of resources to other processes. Therefore, TBB is modified to release its worker threads when idle. TBB owns a static array of worker instances that are assigned

to Posix threads when started. Active workers register in the *tbb::internal::arena* class of TBB to enter a work stealing domain and deregister when they terminate. To comply with the management of worker slots in this class when dynamically allocating and terminating workers, an additional list for sleeping workers is introduced. While the original asleep list stores all workers that are uninitialized and were never started, the new list stores workers that became idle after they were already running. This list is the preferred source of worker instances when new work arises. In order to prevent race conditions when terminating and reallocating worker threads, an additional worker state is introduced. This indicates that a worker thread terminated itself, the corresponding Posix thread is in detached mode, and the worker instance can be reused or cleaned up.

4.3.2 Worker Suspension and Resumption

The developed dynamic resource redistribution mechanism requires all applications to be able to handle resource revocation. As described in section 4.2, an application-specific revocation handler is called using a signaling mechanism which provides information about the event and which execution context was interrupted. The application-specific revocation procedure for TBB is implemented as follows. A pointer to the interrupted worker thread and the corresponding execution context is stored in a global queue that is regularly polled by active workers that run out of work. This queue needs to be synchronized in a lock-free manner, because polling workers might be interrupted during queue access, which can result in deadlocks otherwise. The work-stealing routine of a worker needs to be modified accordingly. When a worker runs out of tasks in its local task queue, it checks whether interrupted workers are available before trying to steal work from remote workers. If interrupted workers are available, the currently active worker migrates the interrupted worker to its local scheduling context and terminates itself in favor of the previously interrupted worker. So, the interrupted worker is restored and continues execution of its task.

CHAPTER 5

Evaluation

The processor allocation and redistribution mechanism developed in this work aims to increase the energy efficiency for multi- and manycore systems. It follows the one-thread-per-core execution model to avoid the overhead of regular context switches and systematically puts unused cores, considering the actual hardware topology, into a specific sleep state to reduce energy consumption and accelerate active cores. The proposed processor allocation mechanism strives to reduce the core allocation latency to motivate user applications for dynamic thread allocation instead of using static thread pools. In addition, this work introduces a resource revocation mechanism allowing asynchronous deprivation of cores that are already allocated to processes but do not meet the global requirements for energy efficiency with their utilization. So, it strives to increase the system global energy efficiency. This chapter evaluates whether the developed concepts are suitable for achieving these goals. Therefore, arising research questions are investigated in individual experiments, including a detailed description of the methodology, expectations, and conclusion each.

5.1 Measuring the Idle State Power Consumption and Wakeup Latency of a Real System

When trying to optimize the computation per watt ratio by reducing energy consumption putting temporary unused processor cores into a sleep state, we need to know how much energy can potentially be saved in which sleep state and what are the costs in wakeup latency when new work becomes available. This influences the decision as to whether it is worthwhile to put cores to sleep or keep them awake for low latency response time. Therefore, this section evaluates the power consumption and wakeup latency depending on the idle state on a real system.

5.1.1 Evaluation System

The processor topology of the computer system used for evaluating the power consumption and wakeup latency is illustrated in figure 5.1. It contains two *Intel Xeon Gold 6238R* processors with 28 cores and 56 hardware threads each, running at 2.2GHz base frequency and a maximum *Turbo-Boost* frequency of 4GHz. The TDP is specified as 165 watts per processor[20].

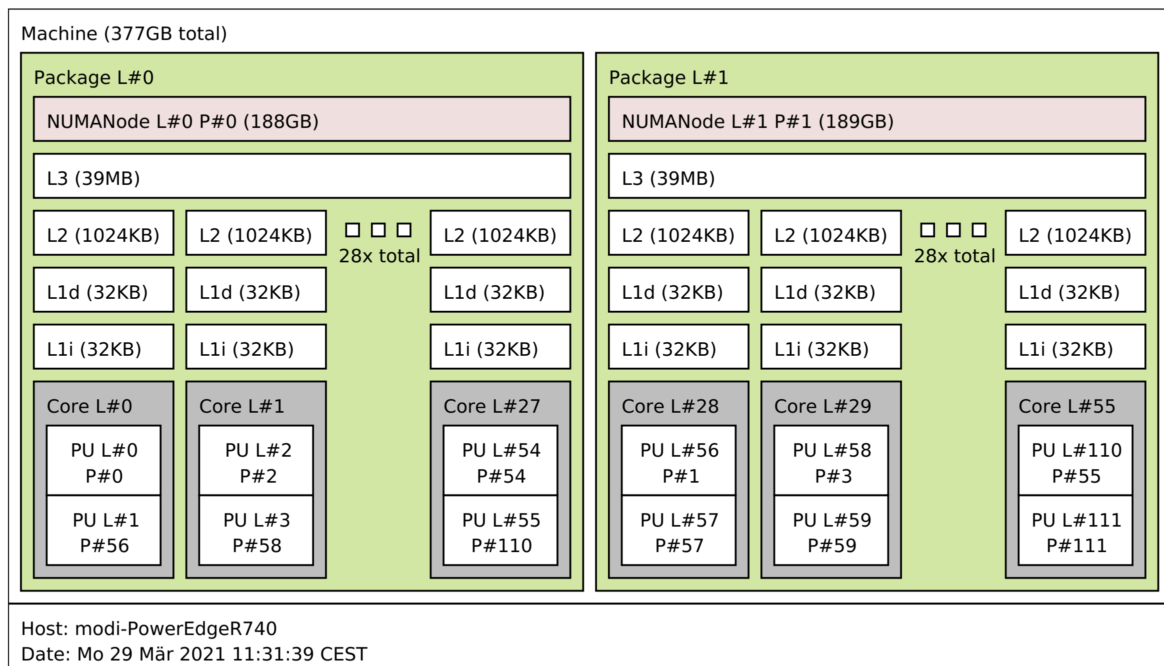


Figure 5.1: Processor topology of the evaluation system (DELL PowerEdge R740 Rack-Server containing two Intel Xeon Gold 6238R[20] processors), generated using hwloc[11].

The *CPUID* instruction exposes that the system supports the *MONITOR/MWAIT* instruction and its power management extension with a fixed monitoring range of 64 bytes, which equals the size of a cache line. According to the monitor/mwait leaf of *CPUID*, the processors offer only two C-states, *C1* and *C3*, with two substates each, that can be entered using the *MWAIT* instruction[18]. This corresponds to the expected available CPU idle states in the Linux kernel. The mentioned processors belong to *Second Generation Intel Xeon Scalable Processors* which are based on the *Cascade Lake* microarchitecture and is referred as family 6 extended model 85 by *CPUID*[20]. This model number is also used for *Skylake X* processors. Hence, the Linux kernel expects the same idle states and lists them as *C1*, *C1E*, and *C6*

but the last is configured as a *C3* hint for *MWAIT*[104]. Additionally, individual wakeup latencies and target residency times are specified as shown in table 5.1.

Name	C-state	C-substate	Exit latency	Target residency
C1	1	0	2 μs	2μs
C1E	1	1	10μs	20μs
C6	3	0	133μs	600μs

Table 5.1: Expected supported C-States, exit latency and target residency time for *Skylake X* and *Cascade Lake* microarchitecture based processors for the *MWAIT* instruction in the Linux Kernel[104]

5.1.2 Accurate Time Measurement on Multi-Core Processors

The measurement of wakeup latencies requires accurate time measurements, which can be achieved by multiple methods. On the given system, the Time Stamp Counters (TSCs) can be used to realize synchronous distributed clocks on all hardware threads, which simplifies the latency measurement. According to *CPUID*, the processors support invariant TSC, which guarantees that the TSC is counting at constant frequency on all cores regardless of the *P-state* and *C-state*. In addition, the TSC frequency can be derived from the core crystal clock frequency which is specified with 25MHz. Using the conversion factor, determined by the TSC leaf of *CPUID*, the TSC frequency results in 2.6GHz. The invariant TSC property does not guarantee the TSC to be synchronized on all hardware threads. Therefore, the TSC offset between the hardware threads needs to be determined and considered in the time measurement. This clock synchronization is done by performing a *Two-Way Message Exchange*[109] via shared memory.

5.1.3 Power Measurement using RAPL

To evaluate energy efficiency, it is essential to be able to measure power consumption. Intel's Running Average Power Limit (RAPL)[18] offers an integrated solution to collect such information and is implemented in recent Intel and AMD processors. Depending on the specific processor, it supports multiple power domains which provide the current energy status and can be configured to define power limits. The *Package (PKG)* RAPL domain refers to the entire socket whereas *Power Plane 0 (PP0)* only concerns all processor cores on this socket and *Power Plane 1 (PP1)* specific devices in the uncore. In addition, there is

an extra RAPL domain monitoring the integrated memory controller that also includes the directly attached *DRAM*. RAPL has been proven to provide accurate energy measurement results compared to external devices[65, 48]. The processors in the evaluation system only deliver values for the *PKG* and *DRAM* domains. Since it is a dual-socket machine, RAPL has to be configured and read for each individual processor socket.

5.1.4 Experimental Measuring of Wakeup Latency

Putting unused processor cores into an idle sleep state comes at the cost of increased wakeup latency when new work arises. This section quantifies those wakeup latencies depending on the C-State.

Procedure After reboot, each hardware thread enters a global barrier. The first hardware thread of the first processor socket acts as the master thread and instructs the other threads to enter a specific C-state by executing *MONITOR/MWAIT* on an individual memory range. The master thread itself does not enter a sleep state. Instead, it waits actively for one second allowing all other threads to enter the target *CC-state*. This prevents the processor from entering a package level C-state higher than *PC0* which would further influence the wakeup latency. Therefore, it is only measured on the first processor socket. For every first hardware thread on each core of the first processor and all available C-states and substates, the wakeup latency is determined using a one-way latency analysis, running the following sequence: The master thread takes a local time stamp using TSC. Thereafter, it writes to the monitored memory range of a specific slave thread and thus wakes it. The slave thread takes a second time stamp. Afterwards, the time stamps are compared. To emphasize the extra wakeup latency, active waiting (polling) on a shared memory variable and *CC0* are measured for comparison.

Expectations It is expected that actively polling on a shared memory region leads to the shortest response time. Periodically checking the memory range and calling *MONITOR/MWAIT* with *CC0* as target sleep state is assumed to be only a little slower than polling since it does not enter a real sleep state but introduces overhead to arm the address monitoring. For every deeper idle sleep state, an increasing wakeup latency is expected. The amount of time required is supposed to be approximately the specified time in the Linux Kernel.

Results Figure 5.2 visualizes the measured wakeup latencies depending on the *CC*-state. Please note the logarithmic scale on the time axis. Actively polling on a shared memory range requires a median of 161.59 nanoseconds, which equals about 137 thousand processor instruction cycles. Calling *MONITOR/MWAIT* with *CC0* increases the latency to 168.64 nanoseconds which meets the expectations, but the variance is bigger and the minimum is lower compared to simple polling. The variance can be explained by the random position in the loop of checking the memory range, setting up the monitor, and calling *MWAIT* when the notification arrives. The lower best-case latency in the *CC0* scenario might be caused by decreased contention on the caches due to the overhead of setting up the address monitoring. Under the use of *CC1* with substate 0 and 1 the median latency equals 858.64 respectively 869.55 nanoseconds which is less than half the latency specified in the Linux kernel for *C1*. Providing *CC3_0* and *CC3_1* as hint to *MWAIT* further increases the wakeup latency to 48.26 and 47.45 microseconds. This equals about one third of the latency specified in the Linux kernel for this configuration.

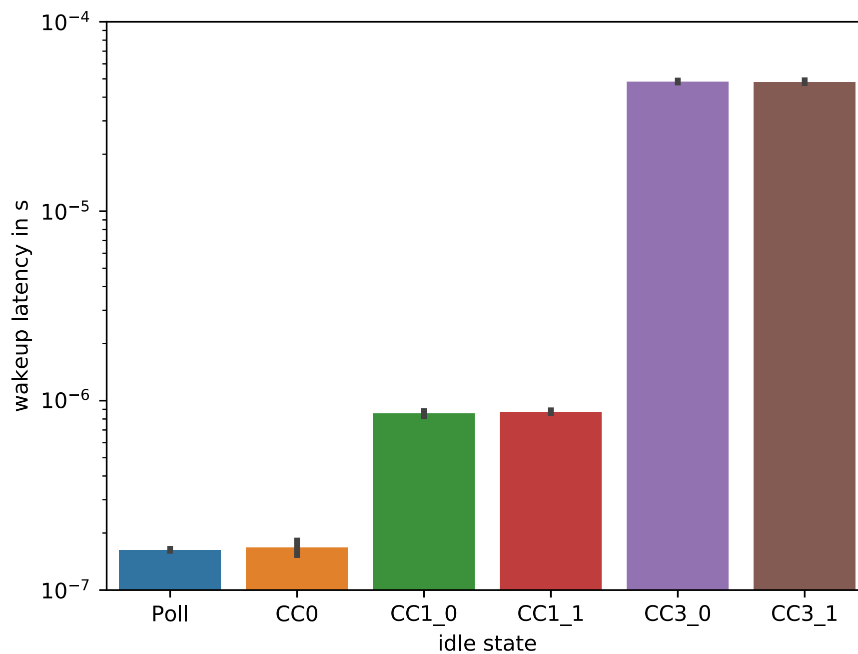


Figure 5.2: Wakeup latency depending on core level idle state and substate using *MONITOR/MWAIT*

5.1.5 Energy Consumption

In order to decide whether it is worthwhile to enter an idle sleep state, with the goal of saving energy, the potential of energy saving needs to be known. Therefore, this section examines the energy consumption of the processor depending on the idle state.

Procedure The current energy status is determined for the entire processor package using the RAPL interface. The power consumption is investigated in one scenario where all cores are asleep and thus the processor can also reach a package level idle sleep state and in a second scenario, where one thread remains active and thus keeps the processor package in *PC0* state. Therefore, the energy status is read and the target idle sleep state is entered using *MONITOR/MWAIT*. After 60 seconds, the energy status is read again. The difference is the amount of consumed energy. The energy divided by the execution time equals the average power consumption.

Expectations The power consumption in *CC0* is expected to roughly correspond to the processors TDP of 165 watts. When entering a deeper core level sleep state, the power is assumed to decrease significantly. Allowing the processor to enter a package level C-state by putting all its cores to sleep should further decrease the energy consumption compared to the scenario where a single thread keeps the package active.

Results Figure 5.3 shows the results. In *CC0* state, the power consumption equals approximately the TDP which meets the expectation. In this case, both packages are in active state but package, configured to put all cores in the desired C-state, consumes less energy than the package where a single thread keeps polling on the TSC in order to wait on the expire of the measurement interval, although the processors are in the same idle state *C0*. This might be caused by different loads caused by polling the TSC and periodically calling *MONITOR/MWAIT*. In addition, the HWP might influence the power consumption. Using *CC1_0* as target C-state reduces the power consumption from 144.97 watts to 54.88 watts for the idle package and from 152.27 watts to 59.12 watts for the package that remains in *PC0*. When applying deeper sleep states, the power consumption decreases further but only in small extent. In *CC3_1*, the power consumption reaches its minimum at 45.02 watts, respectively 55.68 watts. The DRAM power consumption remains, almost unaffected from the idle state, constant between 17.6 watts and 19.7 watts for the active processor, respectively 15.8 watts and 15.9 watt for the idling processor.

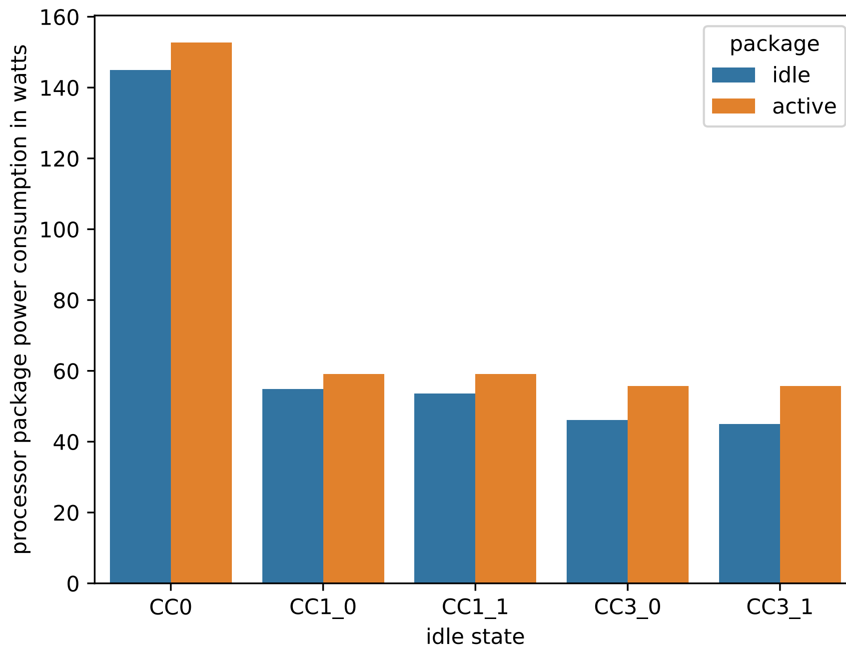


Figure 5.3: Processor package power consumption depending on idle state and substate. Measured using RAPL.

5.1.6 Conclusion

The computation per watt ratio as a measure for energy efficient computing can either be optimized by increasing the computational work or decreasing the required amount of energy. Thereby, the power consumption arises from static and dynamic energy dissipation in CMOS circuits, today's processors are implemented with and which heavily depends on the supply voltage and clock frequency. Power gating, clock gating as well as dynamic frequency and voltage scaling are mechanisms that allow dynamic adjustment of the power consumption according to the current performance requirements. Those hardware mechanisms are partly controllable by the operating system by entering operational performance states (P-states) or idle power saving states (C-states) via ACPI or using processor-specific instructions like *MONITOR/MWAIT*. When a processor is currently unused, deeper C-states bring higher energy savings but at the cost of increasing wakeup latency when needed again. Experiments have proven those expectations and shown that using *CC3* instead of *CC1* sleep state offers only a little benefit in the form of power savings but leads to a significant increase in wakeup latency. Therefore, one can conclude that putting unused processor cores into a sleep state

is a worthy way to dynamically decrease idle power consumption, but suffers from wakeup latency. To compensate this and the power loss of frequent state transitions, a sophisticated OS-level processor management is required.

5.2 Thread Allocation Latency

Due to spatial processor partitioning and exclusive resource allocation, applications are expected to dynamically allocate and deallocate the currently required processing resources instead of continuously keeping a private worker pool of sporadically used processors. Thus, temporally unused processors can be reassigned to other applications or be put to sleep to save energy and, thus, reduce overall power consumption or boost active cores. Users might want to avoid potentially high costs of frequent resource allocation and deallocation. This contradicts the goal of high energy efficiency due to dynamic reallocation. The proposed processor allocation mechanism strives to reduce the core allocation latency to motivate user applications for dynamic thread allocation instead of using static thread pools. Therefore, this section measures the static overhead of creating, starting, and terminating software threads using the developed processor allocation mechanism that is implemented in MyThOS, as described in section 4.1. The results are compared with Linux, because it is one of the most commonly used OSs for parallel computing.

5.2.1 Setup

The benchmark application measures three different latencies. At first, there are direct costs that arise when an application tries to create an additional thread. Therefore, the time until a *pthread_create* function call returns is measured. Usually, when additional threads are requested by an application, new parallel work has been produced and the applications performance depends on a fast start of processing by additional threads. As a consequence, a low latency from calling *pthread_create* until the newly created thread starts execution of productive application code is relevant and, therefore, measured in a second scenario. As a last step, a full cycle from creating a posix thread until it finishes the execution of an empty body function and its termination becomes observable using *pthread_join* is measured. So, the management overhead of creating a thread and waiting until it has processed its work can be determined. All three scenarios are repeated a hundred times to allow statistical evaluation. The results of the proposed processor allocation mechanism implemented in

MyThOS are compared to a stock Linux with kernel version *5.8.0-53-generic*, because Linux and its derivatives are the de facto standard operating systems on the world's most powerful computer systems[107].

The evaluation is performed on a DELL PowerEdge R740 server system containing two *Intel Xeon Gold 6238R* processors with 28 cores and 56 hardware threads each. The system was already introduced in section 5.1.1. This system can be classified as a dual socket, multicore processor system, because each processor owns a single level three cache for all of its cores, making it a single tile architecture. Experiments are performed with *Intel Turbo Boost* feature enabled.

In this experiment, the processor pool balancing thresholds are configured as follows. When the number of free resource in a thread team falls below the lower limit of one hardware thread, more resources are requested from the central processor allocator. When a thread team owns more than four free hardware threads, it returns, if possible, a whole core including its two associated threads back to the central processor allocator. Thus, a thread team should always have at least one free hardware thread in low latency sleep available and no more than four hardware threads per team are prevented to enter a deep sleep state in the central processor allocator. The target sleep state of all hardware threads that are owned by the central processor allocator is *CC3_1* which is the deepest available sleep state of the system. The target sleep state in all thread teams is *CC1_0*, also known as *halt*. So, while hardware threads owned by the central processor allocator are considered cold in terms of wakeup latency and caches, hardware threads owned by the team provide a higher hotness. They have a lower wakup latency and their local caches potentially contain application data from previous assignments which predestines them to serve high frequent allocation requests.

5.2.2 Expectations

On both OSs, the latency until the *pthread_create* function returns is expected to be lower than the latency until the new thread starts its actual work, because it requires waking up another hardware thread. As measured in section 5.1.4, the wakeup latency is between 168 nanoseconds and 48 microseconds, depending on the idle sleep state. Since the thread team is configured to always hold at least one free hardware thread ready in *CC1_0*, the wakeup latency is expected to be around 858 nanoseconds. For all three scenarios, the processor allocation mechanism implemented in MyThOS is not supposed to be slower than Linux, because MyThOS itself consists of a comparatively lightweight structure and, due to the

one-thread-per-core paradigm, there is no need for sophisticated scheduling algorithms like Linux uses when posix threads need to be executed. The third scenario, where a new posix thread is created and joined, is assumed to have the highest latency, because it contains both other scenarios.

5.2.3 Results

Figure 5.4 presents the measured latencies in microseconds of the three scenarios for the developed processor allocation mechanism implemented in MyThOS and compared to Linux. For all three scenarios, the proposed processor allocation mechanism with MyThOS causes a lower latency than Linux, which meets the expectations. The constant median latency value of eight microseconds for all three scenarios under MyThOS can be explained by the overhead for allocating memory and communication to the OS in order to create a new software thread which is the major cost factor compared to the low wakeup latency of the additional hardware thread. Linux causes median latencies of 45 microseconds for the pthread_create scenario, 46 microseconds until the thread starts executing application code, and 47 microseconds from creating until terminating a posix thread. Hence, the measured latencies for all three scenarios are more than five times lower using MyThOS with the developed processor management compared to Linux.

5.2.4 Conclusion

Existing parallel runtime systems try to reduce the overhead of frequent thread allocation by maintaining static thread pools. This behavior hides the dynamics of the application parallelism from the OS and, thus, complicates the resource and power management of the OS. In addition when using spatial processor partition with exclusive allocation, it prevents dynamic reallocation which results in underutilization. The developed processor allocation mechanism that is implemented in MyThOS reduces the allocation latency of a software thread by more than 80 percent compared to Linux. This is not only directly beneficial for performance and energy efficiency but also encourages applications and parallel runtime systems to dynamically allocate and release processing resources.

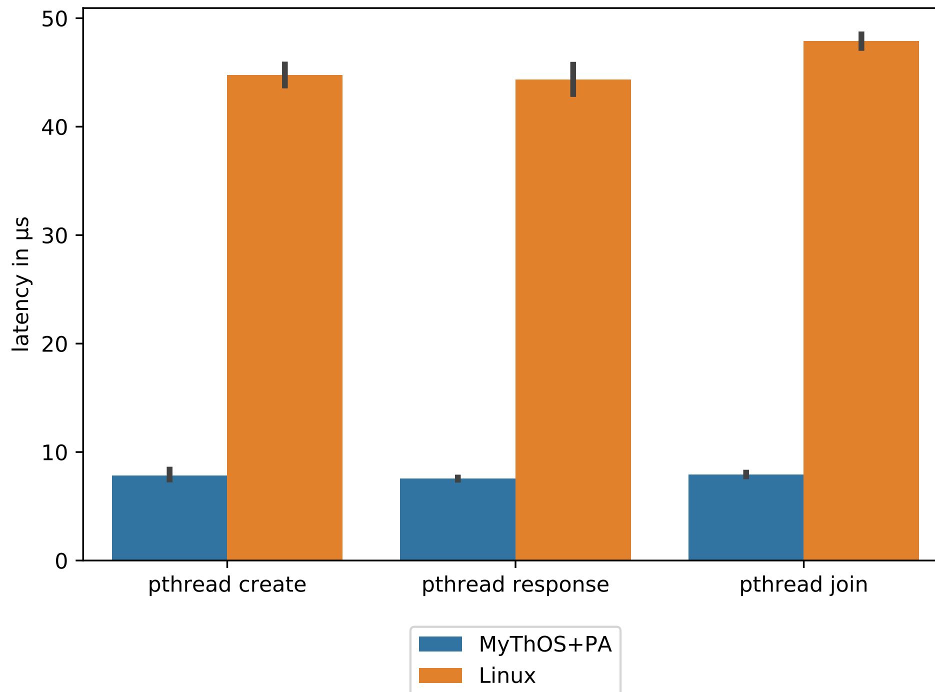


Figure 5.4: Runtime overhead of creating, creating and starting, and creating and joining a posix thread using MyThOS with the developed processor allocation mechanism compared to Linux

5.3 Evaluation of the Energy Consumption and Execution Time on the Example of a Mandelbrot Set Rendering Application

The last experiment examined the direct costs of using the posix thread interface that is implemented as a user abstraction of the developed processor management mechanism and compared it to Linux. In this part, the evaluation focuses on the execution time and energy consumption when running a user application that makes heavy use of the processor allocation mechanism and is highly dynamic in the resource requirements. Therefore, the execution time of this application, that is running on MyThOS combined with the developed processor management mechanism, which requires the application to act resource-aware, is compared to Linux, which does not limit the number of software threads according to the physically available resources. Additionally, the influence of Intel Dynamic Acceleration (IDA)[18] and the target idle sleep state of currently unused cores are investigated.

5.3.1 Mandelbrot Set Benchmark Application

To evaluate the influence of the developed resource management on real applications, a parallel Mandelbrot set benchmark is used. It applies a recursive divide and conquer pattern trying to find sectors of same color by checking a sector's border and splitting it in case of a nonuniform border. Figure 5.5 shows the Mandelbrot set rendering including the frames of the resulting image sectors due to the recursive descent of the employed algorithm.

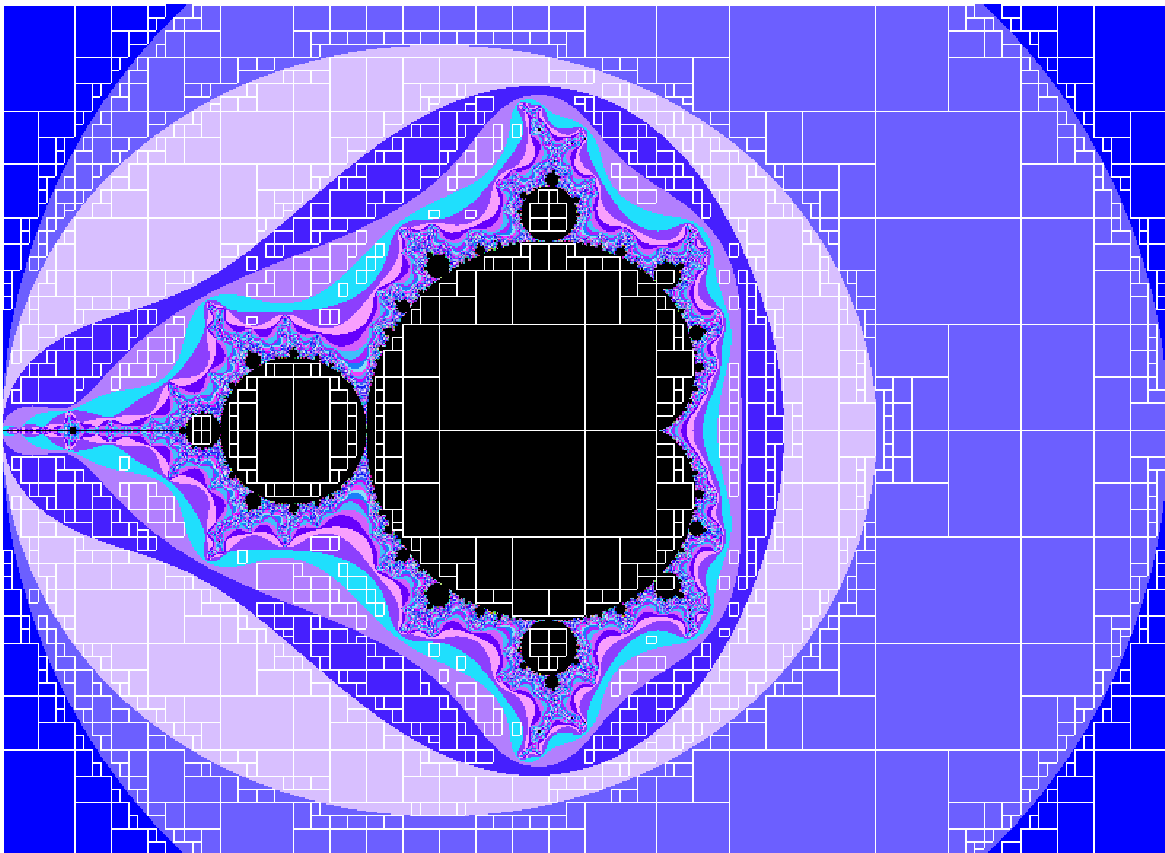


Figure 5.5: Mandelbrot set rendering

Listing 5.1 exposes the algorithm. Based on the configuration of the target image section, an initial image sector is created and the parallel rendering is launched by calling the *render_mandelbrot_sector* function and providing the initial sector to it. If the size of the sector is smaller than a specific lower limit, the algorithm falls back to a serial computation and, thus, iterates over each pixel in the given mandelbrot sector and calculates its color. Otherwise, if the size of the sector is great enough, the algorithm calculates and compares the color of all pixels on the border of the sector. If the border has a uniform color, the

whole sector is expected to have the same pixel color and will be plotted accordingly. In this case, no further computation of the sector is needed. If, on the other hand, the border does not have a uniform color, the algorithm follows a divide and conquer approach. The current sector becomes split up into two smaller subsectors which are then recursively processed using the same function. While one subsector will be handled by the current thread, the other one is delegated to another thread by calling *pthread_create*. If, due to insufficient resources, the creation of the posix thread fails, the current thread processes both subsectors sequentially.

Listing 5.1: Mandelbrot set parallel rendering pseudo-code

```
1  function render_mandelbrot_sector(Sector s)
2    if s < MIN_SECTOR_SIZE then
3      for each pixel in s do
4        calculate color
5        plot pixel
6    else
7      for each pixel in border of s do
8        calculate color
9      if border has uniform color then
10       plot sector
11     else
12       (s0, s1) = split_sector(s)
13       if pthread_create(render_mandel_brot, s0) failed then
14         render_mandelbrot_sector(s0)
15       endif
16       render_mandelbrot_sector(s1)
17     endif
18   endif
```

5.3.2 Setup

The benchmark is configured to render the mandelbrot set in the interval $\{z \in \mathbb{C} \mid Re(z) \in [-1, 1], Im(z) \in [-1, 1]\}$ to a resolution of 4096 by 4096 pixel. Since, saving the resulting mandelbrot set image as a multicolor bitmap would result in massive data movement to the main memory, which would make the application memory-bound, the benchmark viewer only counts the number of pixels inside and outside of the mandelbrot set based on their calculated colors. Thus, the application turns compute-bound through which the influences of processing resource management are emphasized and memory effects like placement decisions

on NUMA machines become negligible. For each required pixel the function $z_{n+1} = z_n^2 + c$, if not diverging earlier, is iterated a thousand times for the associated complex number. For comparison, in the serial scenario, split sectors are not tried to be delegated to new threads but directly be computed by the thread itself. Therefore, *pthread_create* function is not called and no overhead for threading is introduced.

As in the previous experiment, the evaluation is performed on a DELL PowerEdge R740 server system containing two *Intel Xeon Gold 6238R* processors with 28 cores and 56 hardware threads each. The combined energy consumption of both processor packages is measured using RAPL, as described in section 5.1.3.

Power states of idling processor cores are expected to influence the performance of running cores due to *Intel Turbo Boost* as an implementation of IDA[18]. Hence, experiments are performed both with and without the IDA feature enabled. The core clock frequency of the system ranges from 2.2 GHz base frequency to 4GHz maximum turbo boost frequency. The serial and parallel execution time depending on the activation of IDA of the proposed processor allocation mechanism implemented in MyThOS are compared to a stock Linux with kernel version *5.8.0-53-generic*, because Linux and its derivatives are the de facto standard operating systems on the world's most powerful computer systems[107].

In order to provide evidence about the execution time and energy consumption depending on the number of employed hardware resources, the thread team is limited in multiple steps from a single to all available 112 hardware treads for the scalability scenario. Thereby, the target idle sleep state of cores that are currently owned by the central processor allocator and cores that are bound to a thread team will be varied in order to inspect the influence of different latency and energy characteristics.

5.3.3 Expectations

The serial mandelbrot set calculation, where *ptread_create* is not called, is assumed to produce approximately the same execution time on MyThOS with the processor management compared to Linux, because no OS functionality is used. Additionally, in the serial case, activating IDA is expected to reduce the execution time at most by the ratio between the base clock frequency and the maximum turbo boost frequency. This speedup is considered to be significant, since the application is compute-bound and all other cores are supposed to be in an idle sleep state, saving energy and, therefore, boosting the active core. Running the application in parallel, trying to spawn another posix thread on each sector split, MyThOS

in combination with the proposed processor management is expected to require less execution time than Linux, because the resource management limits the number of posix threads that can be spawned to the number of available resources, while Linux is confronted with the overhead of a not directly limited number of threads. When scaling the thread limit of the thread team, that represents the benchmark process under MyThOS, the execution time is assumed to lower sublinearly with an increasing number of processors, but might experience a knee in the speedup curve due to the growing management and synchronization overhead in the runtime system. On the one hand, increasing the number of active cores also increases the power consumption, because those cores cannot enter an idle sleep state and, therefore, do not save energy. However, using more cores for processing the fixed sized application is expected to shorten the execution time. So, when the application finishes execution earlier, the increased power consumption is limited to a short interval, which still might reduce the total energy required to execute the benchmark. Whether it is more energy efficient to use a high or low number of processing resources to process the benchmark application is hardly predictable but is clarified by the measurements.

Putting unused cores into deep sleep promises low energy consumption at the costs of high wakeup latency, while low sleep states offer a lower wakeup latency but high energy consumption. Configuring the target idle state to deep sleep state for all cores owned by the central processor allocator combined with halt state for cores owned by a thread team promises a good trade-off between wakeup latency and energy savings, that can be used to boost active cores. Therefore, this is used as the default configuration and compared with other combinations.

5.3.4 Results

This section presents and discusses the measurements of the mentioned experiments. At first, the execution time comparison to Linux in dependence of the serial and parallel execution and IDA feature is examined. Afterwards, the scalability with increasing number of employed cores and multiple combinations of idle sleep states is investigated. The last part inspects the influence on the energy consumption for the fixed application and for a fixed time interval.

Execution Time Comparison to Linux Figure 5.6 shows the execution time of the mandelbrot set benchmark application running serial and parallel on MyThOS with the developed processor management and with Linux. Additionally, the influence of IDA is measured.

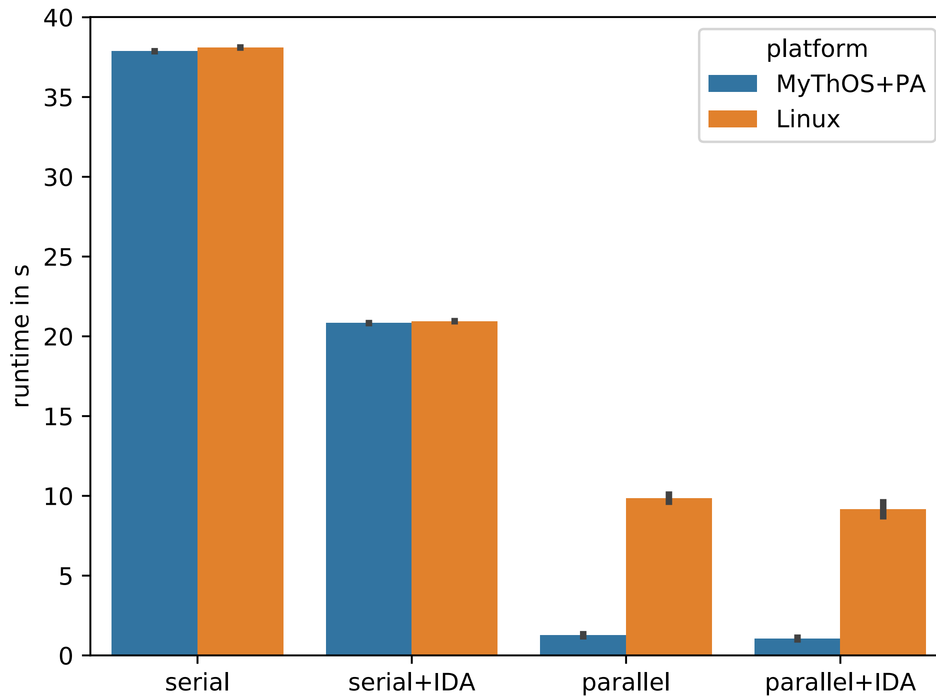


Figure 5.6: Mandelbrot set benchmark execution time comparison Linux versus MyThOS with the proposed processor allocation mechanism in dependency of activated IDA

In the serial scenario without IDA enabled, the execution time of MyThOS with the processor management and Linux are with a median of 37.87 respectively, 38.11 seconds, similar. This meets the expectation. The slightly lower execution time on MyThOS can be explained with the simple memory management in MyThOS user space that eagerly maps all available memory to the address space at startup and, therefore, avoids the overhead of lazily mapping memory pages. Additionally, Linux might interrupt the active core in order to process kernel tasks or other user processes while MyThOS provides exclusive access to the core. With the IDA feature enabled, the serial execution time decreases to 20.83 seconds for MyThOS and 20.95 seconds for Linux which equals a speedup by factor 1.82 in both cases. This correlates exactly with the increased maximum core clock frequency from 2.2GHz to 4GHz due to turbo boost. From this observation, one can derive that the application can take a maximum benefit from the frequency scaling. This is caused by the fact that the application is designed to be compute-bound and, thus, is not limited by memory access latency.

When running the parallel version of the benchmark application, each time a mandelbrot set sector becomes split, it tries to create a new pthread for the processing of one subsector. Without IDA, on MyThOS, including the processor management, a benchmark iteration requires a median of 1.27 seconds which results in a speedup of 29.82 compared to the serial case. In the same scenario, the benchmark required 9.84 seconds running on Linux. This means a speedup of 3.87 but is still 7.75 times slower than MyThOS. The deviation of the execution time measurements is mainly caused by the resource aware approach of the proposed processor management mechanism in MyThOS which is based on the one-thread-per-core paradigm and, thus, prevents the application from creating more software threads than physical processor units available. Linux, on the other hand, does not directly limit the number of software threads. So, more and more software threads are created, regardless of the current system state and the overhead for management and rescheduling which causes the major part of the execution time. Enabling the IDA feature further reduces the execution time to 1.08 seconds for MyThOS and 9.22 for Linux which leads to a speedup of 1.18, respectively 1.07 compared to the parallel execution without IDA. From that one can conclude that, for the given scenario, IDA has less impact on execution time when all cores are active compared to when many cores entered an deep idle sleep state.

Scalability Depending on Target Idle Sleep State The thread team kernel object, as a part of the processor management mechanism in MyThOS, offers the functionality to configure an upper limit of hardware threads that can be allocated by the corresponding user process. So, it allows the user to restrict its resource footprint without keeping books itself. This mechanism is employed to measure the application scalability at runtime. Additionally, multiple combinations of target idle sleep states for the central processor allocator and the thread team are evaluated. IDA is activated. The results are shown in figure 5.7.

In this scenario, limiting the number of threads to one is not equal to the serial run of the previous experiment, because, due to the thread restriction, the application fails when trying to create additional software threads while the serial version does not even try to create them and directly processes all mandelbrot subsectors itself. Consequently, the execution time required for a mandelbrot benchmark iteration increases from a median of 20.83 seconds of the serial version to 21.44 seconds of the parallel version with the number of threads limited to one. Therefore, continuously trying and failing to create additional software threads increases the execution time by factor 1.03. In both cases, the target idle sleep state of the central processor allocator is configured to deep sleep (*CC3_1*) and the thread team to halt (*CC1_0*). Keeping all cores spinning in an active state increases the required execution

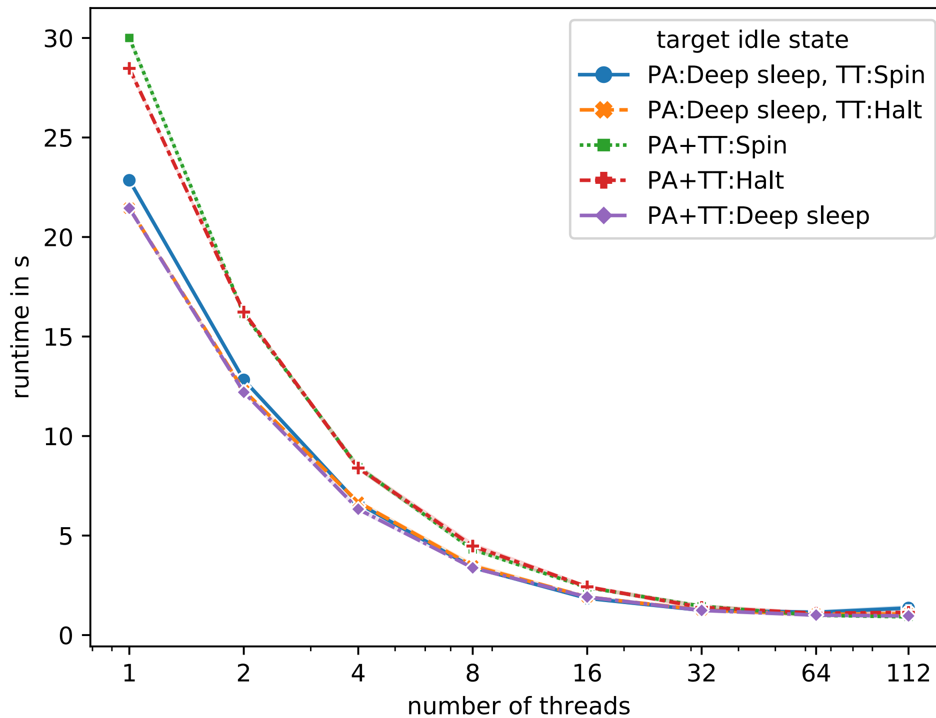


Figure 5.7: Execution time of mandelbrot set benchmark on MyThOS with the processor allocation mechanism depending on thread limit and target sleep modes with active IDA

time by factor 1.4 to 29.98 seconds. Since only a single hardware thread is executing the benchmark code, all other cores are wasting energy that the benchmark thread lacks for boosting its frequency. Similarly, using halt as a global target sleep state causes idling threads not to save enough energy to for maximum boosting of the active thread. So, a median execution time of 28.47 seconds is required. A global target idle state configured to deep sleep produces the same execution time of 21.44 seconds as the default combination of deep sleep and halt. This is caused by the fact that in both cases most cores are owned by the central processor allocator and, therefore, entered the deep sleep state. If, on the other hand, the thread team is configured to keep its idle core spinning, the execution time increases to 22.85 seconds due to energy wastage.

When the thread limit of the team is increased, the execution time required for the benchmark application decreases as well as the absolute differences in execution time between the sleep state configurations. This meets the expectation. The lowest execution time of 0.93 seconds

is reached using all available 112 hardware threads and keeping all idle threads actively spinning. This supports the thesis that, if the system load is high and cores are only used for a short period of time, it is more time efficient to keep the cores active instead of putting them into an optimized sleep state. On the other hand, using deep sleep as target idle state leads to an 1.04 times higher execution time of 0.97 seconds in median. One can argue that all cores are nearly permanently in use and, thus, do mostly not enter the full deep sleep state. The default combination of deep sleep and halt produces a execution time of 1.08 seconds, which is 1.16 times slower than spinning and 1.11 times slower compared to using a global deep sleep. Consequently, for the given scenario, using deep sleep as target idle state for both the central processor allocator and thread team, offers the best trade-off between energy savings and wakeup latency and, consequently, produces almost the lowest execution time for all numbers of used threads. This phenomenon is suspected to be restricted to the characteristics of the benchmark application where a constant group of threads is nearly permanently occupied while all other cores remain idle and stay in their desired sleep state. Frequent switching of the target sleep state when balancing threads between processor pools requires the cores to be woken up and introduces runtime overhead. Moreover, one must take into account that the provided target sleep state is only seen as a hint by the CPU and the actually selected state may differ.

Energy Scalability Depending on Target Idle Sleep State While the previous section only considered execution time as a measure for efficiency of the processor management mechanism in MyThOS, the following part focuses on energy consumption. Thereby, the same experimental setup is used and IDA is active. The total energy consumed by both processor packages in the system for the fixed benchmark iteration is shown in figure 5.8.

Deep sleep idle states promise lower power consumption compared to lower sleep or active states. The combination of the high power consumption and long execution time when keeping all idle threads actively spinning, causes the highest energy consumption of 9.51 kilojoule when using only a single thread for the application. In the single thread scenario, the lowest energy consumption of 2.37 kilojoule is reached when configuring the target idle sleep state globally to deep sleep or to a combination of deep sleep and halt. This meets the expectation and means an energy reduction of more than 75 percent due to less required power of the idle threads and lower execution time due to increased performance boost. Generally, increasing the thread limit in the team decreases the energy required to process a benchmark iteration, but is mainly caused by the reduced execution time. The lowest energy consumption is measured using 64 threads and deep sleep as the target sleep state for all

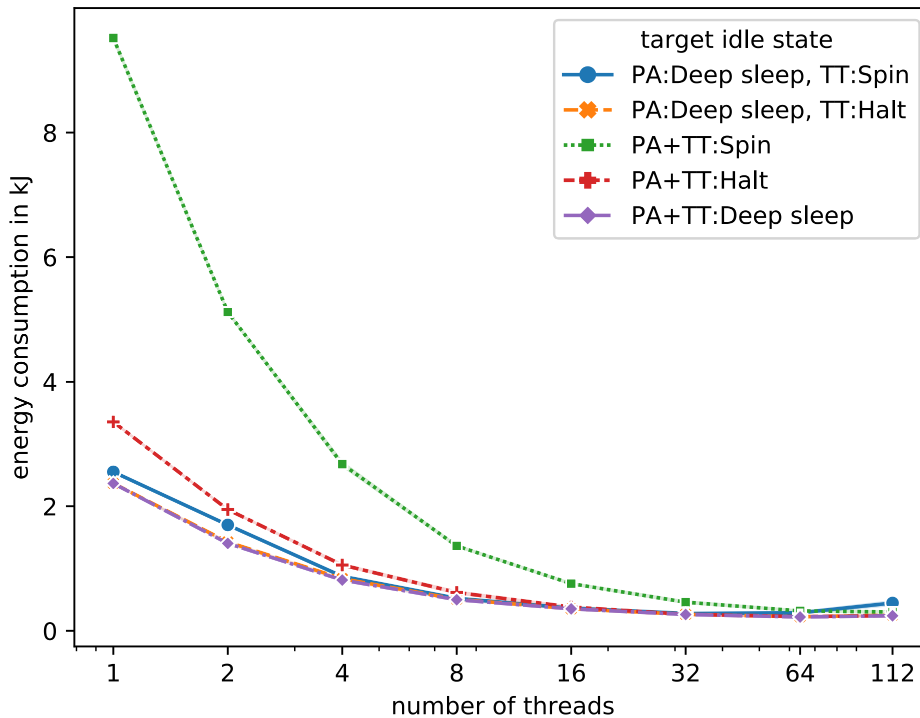


Figure 5.8: Energy consumption of both processor packages for Mandelbrot set benchmark on MyThOS with the processor allocation mechanism depending on the thread limit and target sleep modes with activated IDA

threads. In this case, it requires a median of 0.22 kilojoule to process a benchmark iteration. Hence, the number of threads assigned to this benchmark to reach the lowest execution time does not provide the best energy efficiency. When using all available 112 hardware threads of the system, the energy consumption increases slightly for all sleep state configurations.

In this experiment, only the energy consumption over the execution time of a single benchmark iteration, that is specific for the individual configuration, is measured. Following the argumentation that, directly after the completion of an application, the system can start to process other applications or be fully shut down to save energy, the energy consumption over the execution time delivers a suitable measure for energy efficiency. When, on the other hand, the system is supposed to stay awake in order to be available for spontaneously arising tasks, the system can just enter a deep sleep state instead of shutting down. Since processors in deep sleep state still consume energy, the energy consumption has to be scaled to a fixed time interval to compare the energy efficiency of multiple configurations leading

to individual execution times. So, one can decide whether it is more energy efficient to quickly finish execution using many threads and then enter deep sleep mode or using less threads for processing the application while the other threads are kept in deep sleep for a longer execution time. Since the energy consumption over a fixed interval that includes the benchmark execution time of each configuration is not directly measured, the fixed interval specified to the longest execution time of about 3.02 seconds and the deep sleep power of 45.02 watts, as measured in section 5.1.5, is added for the remaining time for each individual configuration. This scaled energy consumption is shown in figure 5.9.

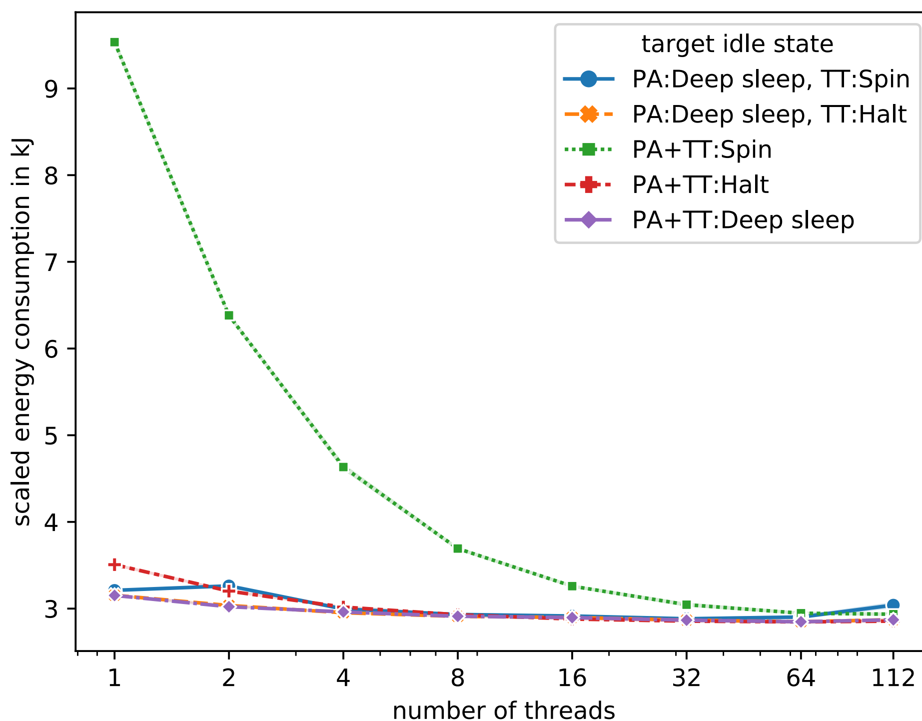


Figure 5.9: Mandelbrot set benchmark energy consumption of both processor packages on MyThOS with the processor allocation mechanism depending on the thread limit and target sleep modes scaled to fixed execution time and with IDA activated

The adapted energy consumption that considers the measured deep sleep idle power to fill up the execution time to the fixed time interval, emphasizes the importance of putting idle cores to sleep. In the single threaded scenario, the energy consumption for the deep sleep configuration, respectively, the deep sleep and halt combination, increase by a factor of 1.33 to 3.15 kilojoule. The highest energy consumption of the idle spinning configuration stays

constant, because this is used as reference case for the fixed interval. The lowest energy consumption over the fixed interval of 2.85 kilojoule is produced using 64 threads and the target idle sleep state configured to either deep sleep, halt, or a combination of both. Thereby, less than eight percent of the energy is used for actual computation while the rest is consumed for deep sleep idle until the fixed interval ends. From this, one can derive that, for the given scenario and application, it is generally slightly more energy efficient to employ a high number of threads and quickly enter system wide deep sleep than using less threads over a larger period of time, but it is most important to put unused cores into a sleep mode.

5.3.5 Conclusion

The developed resource management in MyThOS limits the number of allocated software threads to the number of physically available hardware threads. This requires the application to act resource-aware and thus being able to handle denied processing resource requests. In combination with the reduced thread allocation latency, this results in more than seven times faster execution of the parallel mandelbrot set rendering application than Linux, which does not limit the number of parallel software threads and, therefore, causes massive overhead for thread allocation and rescheduling.

The experiments have shown that, for the compute-bound application under study, it is more energy efficient to process the application using a high number of active processor cores and, thus, finish execution earlier than using only a single hardware thread and putting others to sleep. This trend is still valid when considering a fixed time interval instead of a fixed application size and putting all cores into a deep sleep state after termination. The experiments also have shown that a target sleep state for all threads configured to *CC3_1* (*deep sleep*), in generally produces the lowest execution time and the best energy efficiency. The combination of deep sleep for all threads owned by the central processor allocator and *halt* as a target sleep state for all threads of the team, is slightly less efficient in execution time and energy but is still expected to be a good default configuration, especially when running real world applications that do not permanently make excessive use of the resource management mechanism. From the results, one can derive that a proper use of processor idle sleep states has a major impact on the energy efficiency as well as the performance, especially when dynamic performance boosting is activated and not all cores are permanently in use.

5.4 Automated Processing Resource Allocation in the Parallel Runtime System

The mandelbrot set benchmark application of the previous experiment directly uses the extended posix thread interface to express its parallelism and is, consequently, itself responsible for handling failed resource allocation requests and thread synchronization. Parallel runtime systems ease the development of parallel applications, because they implement the basic functionality of the parallel execution model, for example, taskification, synchronization, work distribution, and work balancing. As examined in section 3.5, this work proposes to shift the dynamic processing resource handling into the parallel runtime system as well, to disburden the application programmer from the manual implementation. This strives to increase the productivity by reducing the code complexity and thus susceptibility to errors.

Section 4.3 described the exemplary integration of automated processing resource allocation into Intel Threading Building Blocks (TBB) using adaptive work stealing. Instead of directly creating and programming software threads, TBB offers multiple mechanisms to specify fine-grain tasks, which are in dependable work packages that are scheduled using work-stealing. This allows for a high-level abstraction of parallel program sections. Even if using a work stealing task scheduler potentially reduces the number of thread allocations, the required task creation and task scheduling still introduces overhead. Hence, there is a tradeoff between frequent thread allocation and overhead of the task-parallel runtime system, which directly influences the application performance and energy efficiency. Thus, this experiment targets the question of how the usage of this task-parallel runtime system with automated processing resource allocation affects the performance and energy efficiency compared to manual thread allocation.

5.4.1 Setup

To emphasize the costs of dynamic multi-threading or task creation and task scheduling, this experiment uses the mandelbrot set rendering benchmark application from the previous experiment, which is described in section 5.3.1. However, the mandelbrot set benchmark application needs to be modified, to use the task interface of TBB instead of plain posix threads. Thus, the call of *pthread_create()* and its error handling presented in line 13 to 15 of listing 5.1 are replaced by *tbb::task_group::run()* to create a new task in the TBB task group. The handling of unsuccessful thread creation is no longer required, because the task

creation cannot fail and tasks are automatically distributed to available worker threads by the work stealing scheduler.

Equally to the previous experiment, the benchmark is configured to render the mandelbrot set in the interval $\{z \in \mathbb{C} \mid Re(z) \in [-1, 1], Im(z) \in [-1, 1]\}$ to a resolution of 4096 by 4096 pixel. Furthermore, only the number of pixels inside and outside of the mandelbrot set are counted, based on their calculated colors. Thus, the application turns compute-bound through which the influences of processing resource management are emphasized and memory effects like placement decisions on NUMA machines become negligible. For each required pixel the function $z_{n+1} = z_n^2 + c$, if not diverging earlier, is iterated a thousand times for the associated complex number.

As in the previous experiments, the evaluation is performed on a DELL PowerEdge R740 server system containing two *Intel Xeon Gold 6238R* processors with 28 cores and 56 hardware threads each. The combined energy consumption of both processor packages is measured using RAPL, as described in section 5.1.3.

The execution time and energy consumption of the TBB and posix thread mandelbrot set rendering variants are compared to each other. Therefore, they are both executed on MyThOS with the proposed processor allocation mechanism. The IDA feature is activated. In order to provide evidence about the execution time and energy consumption depending on the number of employed hardware resources, the thread team is limited in multiple steps from a single to all available 112 hardware treads for the scalability scenario.

In this experiment, the processor pool balancing thresholds are configured as follows. When the number of free resource in a thread team falls below the lower limit of one hardware thread, more resources are requested from the central processor allocator. When a thread team owns more than four free hardware threads, it returns, if possible, a whole core including its two associated threads back to the central processor allocator. Thus, a thread team should always have at least one free hardware thread in low latency sleep available and no more than four hardware threads per team are prevented to enter a deep sleep state in the central processor allocator. The target idle sleep states are configured to *deep sleep* for all threads owned by the central processor allocator and *halt* as a target sleep state for all idle threads assigned to a team. So, while hardware threads owned by the central processor allocator are considered cold in terms of wakeup latency and caches, hardware threads owned by the team provide a higher hotness. They have a lower wakup latency and their local caches potentially contain application data from previous assignments which predestines them to serve high frequent allocation requests.

5.4.2 Expectations

Creating a task each time a subsector of the mandelbrot set needs to be rendered instead of trying to create another posix thread reduces the overhead spent on thread creation. Nevertheless, creating a task requires memory allocation and the task scheduler introduces additional overhead. However, using the tasking framework of TBB is not expected to result in much greater but in a slightly lower execution time, because the posix thread allocation requires memory allocation as well. Additionally, creating a new posix thread requires interaction with the resource management in the OS and failed thread allocation requests may lead to work imbalances due to the recursive execution of both mandelbrot subsectors by the current thread itself. The previous experiment has shown that the energy consumption for a fixed sized application decreases with a lower execution time. Hence, the required energy for the execution of the TBB mandelbrot set application variant is expected to be equal or lower compared to the posix thread variant.

5.4.3 Results

Figure 5.10 shows the execution time for the mandelbrot set benchmark depending on the number of hardware threads assigned for both the posix thread and the TBB implementations. Using TBB tasks for the parallelization results in a general reduction of the execution time compared to posix threads. This meets the expectation. For the serial case, where no additional threads can be created, the execution time decreases by 3.2 percent to 20.794 seconds. Using a high number of hardware threads increases the execution time reduction, so that when assigning all available 112 threads to the application, the TBB variant decreases the execution time by 58.8 percent to 0.418 seconds compared to the posix thread variant.

The energy consumption per mandelbrot set benchmark run depending on the hardware thread limit and the parallelization approach is presented in figure 5.11. Similar to the execution time, using the TBB implementation generally reduces the energy consumption compared to posix threads. This meets the expectation. Hence, when the application is limited to a single hardware thread, the TBB implementation decreases the energy consumption by 3.8 percent to 2.305 kilojoule. Furthermore, a high number of assigned hardware threads intensifies the differences in energy consumption. If all 112 hardware threads are available to the application, the TBB implementation reduces the required energy by 45.4 percent to 0.141 kilojoule compared to the posix thread variant.

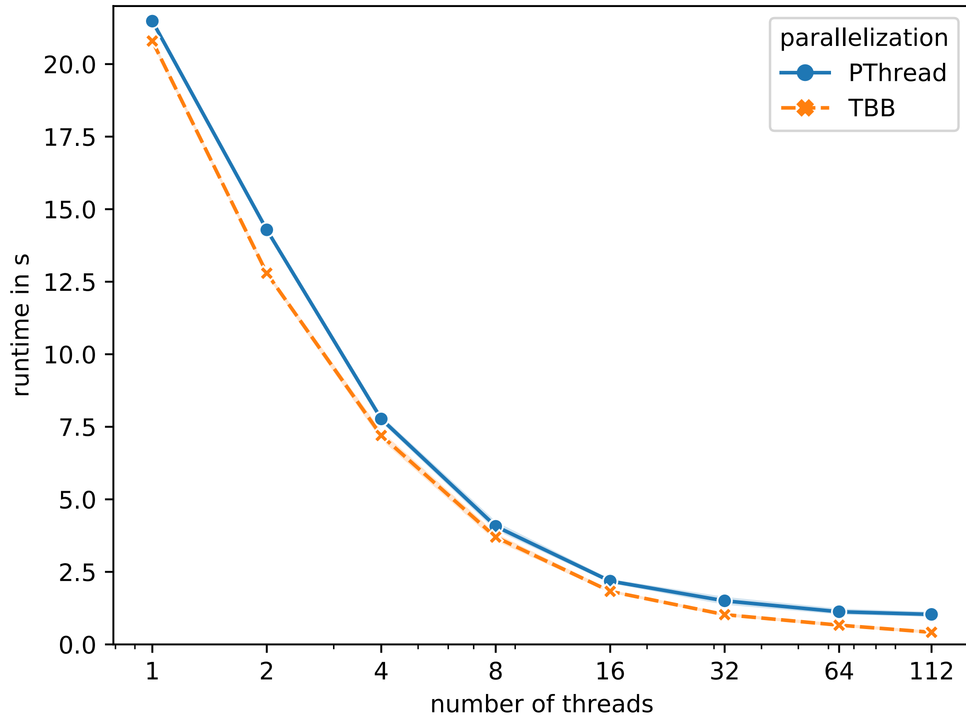


Figure 5.10: Execution time of the mandelbrot set benchmark on MyThOS with the processor allocation mechanism depending on thread limit and parallelization approach

5.4.4 Conclusion

Existing parallel runtime systems facilitate the development of parallel applications, because they implement the basic functionality of the parallel execution model. This work proposes to shift the dynamic processing resource handling into the parallel runtime system as well, to disburden the application programmer from the manual implementation. This strives to increase the productivity by reducing the code complexity and thus susceptibility to errors. On the other hand, using a task-based runtime system instead of directly programming posix threads affects the execution behavior and thus performance and energy efficiency because it reduces the number of thread allocations but generates overhead for task creation and scheduling. Since this work strives to maximize energy efficiency, it is important that the migration from manual thread allocation to the usage of a task-parallel runtime system does not degrade performance and energy consumption. Therefore, this experiment evaluated the question of how the usage of this task-parallel runtime system with automated processing

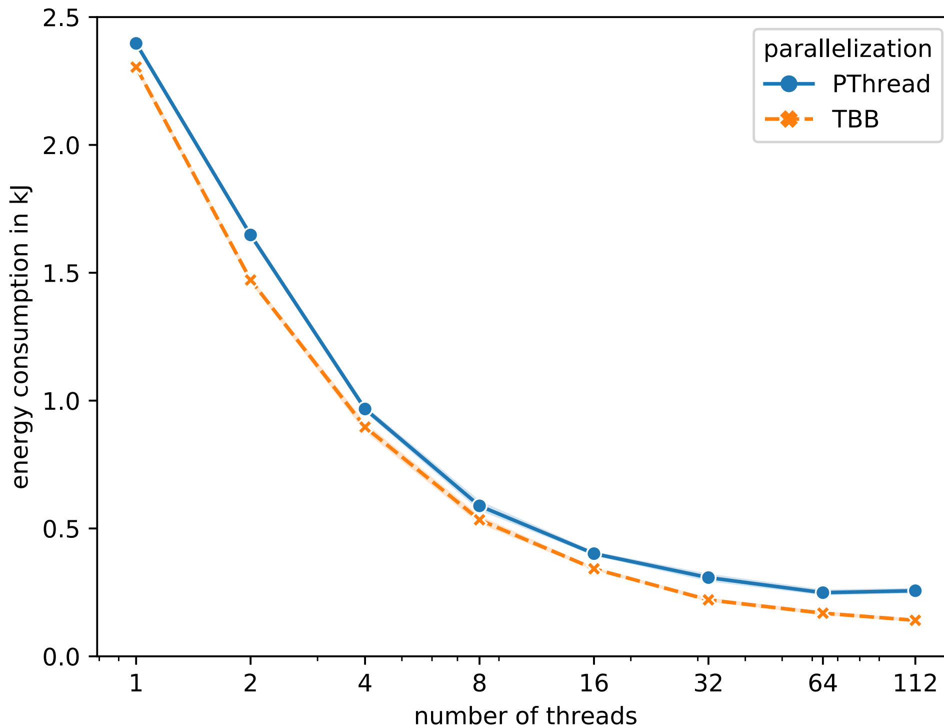


Figure 5.11: Mandelbrot set benchmark energy consumption of both processor packages on MyThOS with the processor allocation mechanism depending on the thread limit and parallelization approach

resource allocation affects the performance and energy efficiency compared to manual thread allocation.

The results of this experiment have shown that using TBB as a task-parallel runtime system leads to a reduction in execution time from 3.2 percent up to 58.8 percent and a decrease of energy consumption from 3.8 percent up to 45.4 percent depending on the number of assigned processing units compared to the manual thread allocation. From this one can conclude that the usage of a task-parallel runtime system is not only not harmful but beneficial for energy efficiency and represents thus a suitable basement for automatic processing resource handling.

The synthetic mandelbrot set benchmark application that was used in this experiment makes excessive use of fine-grain task creation and thus emphasizes the costs for task management, but does not allow for dynamic allocation optimizations because it operates in only a single program phase with a high degree of parallelism until it finishes execution. Therefore,

the following experiment focuses on the influence of dynamic thread allocation on a real world application that runs in alternating phases with different resource requirements in the parallelism profile.

5.5 Impact of Dynamic Resource Allocation in Parallel Runtime Systems on Performance and Energy Efficiency

The previous experiment investigated the question of how the usage of a task-parallel runtime system with automatic processing resource allocation affects performance and energy consumption compared to manual thread allocation. It has shown that it leads to a significant reduction of execution time and energy consumption. Additionally, it reduces code complexity and thus increases productivity. Hence, it was concluded that using a task-parallel runtime system with automated processing resource allocation is a suitable approach to disburden the programmer from manual thread allocation. However, the previous experiment used a synthetic mandelbrot benchmark that makes excessive use of fine-grain tasks and thus only emphasizes the direct costs of task management, but does not allow dynamic allocation optimizations by resource management in the OS. This is caused by the fact that the application consists only of a single program phase with massive parallelism due to its recursive fork-join pattern. In order to demonstrate the benefits of the developed resource and dark silicon management, this experiment evaluates the influence of dynamic resource allocation on a real world application that operates in multiple phases with different parallelism.

This experiment uses One-Dimensional Turbulence Large Eddy Simulation (ODTLES)[42, 43, 64] as a benchmark application. One-Dimensional Turbulence (ODT)[63, 76] is a stochastic turbulence model that models the transport of turbulent advection in 3D flows by stochastic transformations of scalar 1D profiles. ODTLES is the application of ODT as a sub-grid model to close unresolved small-scale dynamics of the Extended Large Eddy Simulation (XLES)[42]. This benchmark application is used because it represents an example for a typical scientific numerical simulation, that is not perfectly parallelized and thus operates in phases with different parallelism. Therefore, it is expected to offer potential for dynamic optimization of resource allocation.

5.5.1 Setup

The experiment is arranged as follows.

ODTLES Benchmark Application In this experiment, ODTLES[42, 43, 64] is used as a real-world benchmark application. It implements parallelization using a combination of Message Passing Interface (MPI)[14] and Open Multi-Processing (OpenMP)[37, 105]. This work focuses on resource management at the level of a single shared memory node. Thus, the parallelization using only OpenMP without MPI is sufficient, because MPI is only required for communication between individual shared memory nodes within a cluster computer. In addition, the dynamic resource handling developed in this work is exemplarily integrated into TBB only, but not into OpenMP. Hence, the implementation of ODTLES has been ported from OpenMP to TBB. ODTLES expresses its parallelism using only OpenMP *parallel for* loops, which are directly replaceable by TBB, because it supports the same abstraction.

Parallelism Profile The ODTLES application is expected to operate in multiple program phases with varying degree of parallelism which potentially allows for dynamic allocation optimization. To prove this expectation, the parallelism profile of this application is measured in this experiment. Therefore, the TBB runtime system is instrumentalized and extended by a tracing component that logs the activity of the workers. So, a histogram of the parallelism in form of worker activity is determined.

Dynamic Worker Pool As examined in section 3.5, this work proposes to employ adaptive work stealing in the runtime system, to dynamically negotiate the allocation of resources with the processor management based on the actual available work and the load of the system. It promises to be a reasonable approach to disburden the application programmer from manual processing resource allocation while still providing information about the application's dynamics in parallelism to the processor management. Hence, the TBB runtime system is modified to allocate workers, including processing units, on demand, and terminate idle workers so that the corresponding processing units are released to the resource management again.

Evaluation System As in the previous experiments, the evaluation is performed on a DELL PowerEdge R740 server system containing two *Intel Xeon Gold 6238R* processors with 28 cores and 56 hardware threads each. The combined energy consumption of both processor packages is measured using RAPL, as described in section 5.1.3.

The execution time and energy consumption of the ODTLES benchmark application using TBB once with static worker pool and dynamic worker pool are compared to each other. Therefore, both are executed at MyThOS with the proposed processor allocation mechanism. The IDA feature is activated. To provide evidence on the execution time and energy consumption depending on the number of employed hardware resources, the thread team is limited in multiple steps from a single to all available 112 hardware treads for the scalability scenario.

In this experiment, the processor pool balancing thresholds are configured as follows. When the number of free resource in a thread team falls below the lower limit of one hardware thread, more resources are requested from the central processor allocator. When a thread team owns more than four free hardware threads, it returns, if possible, a whole core including its two associated threads back to the central processor allocator. Thus, a thread team should always have at least one free hardware thread in low latency sleep available and no more than four hardware threads per team are prevented to enter a deep sleep state in the central processor allocator. The target idle sleep states are configured to *deep sleep* for all threads owned by the central processor allocator and *halt* as the target sleep state for all idle threads assigned to a team. So, while hardware threads owned by the central processor allocator are considered cold in terms of wakeup latency and caches, hardware threads owned by the team provide a higher hotness. They have a lower wakup latency and their local caches potentially contain application data from previous assignments which predestines them to serve high frequent allocation requests.

5.5.2 Expectations

ODTLES is expected to operate in multiple program phases with varying degrees of parallelism. This behavior should be visible in the parallelism profile.

Due to its parallelism, ODTLES is expected to benefit from multiple processing units. So, the required execution time is supposed to decrease compared to the sequential execution. Nevertheless, the application includes sequential parts which limit its scalability. Thus, the speedup is expected to scale sub-linearly with the number of processing resources and

might experience a sweet spot after which the execution time starts to increase again. As experienced in the previous experiments, the energy consumption required for an application mainly depends on its execution time. However, actively using a high number of processing units leads to a high power consumption and if those processing units are not able to further decrease execution time, caused by the limited application scalability, the energy efficiency decreases.

When using a static worker pool, worker threads are only allocated at the first parallel program section, put to sleep when out of work, and terminated when the program finishes. As examined in section 2.4, this scheme is typically used to avoid the overhead of frequent software thread allocations. The sleeping of idle workers in TBB is implemented using a FUTEX[38] mechanism. In MyThOS, processing units that are bound to a blocked software thread, which is the case if it is waiting on a FUTEX, are configured to enter *HALT* idle state, because it is unknown for how long the software thread will be blocked. Therefore, those processing units, that are bound on sleeping worker threads, are hindered from entering a deep sleep idle state which decreases the energy efficiency during long sleeping intervals. In contrast to this, the dynamic worker pool implementation releases idling worker threads, and therefore the corresponding processing units, and reallocates them when new work is available. This tends to increase the number of worker thread allocations during the application execution but allows for improved dark silicon management in the OS, because in this way the dynamics in parallelization of the application can be distinguished from short-term blocking of threads due to synchronization using a FUTEX. Although the dynamic worker pool variant is supposed to increase the overhead for worker thread allocation, it is expected to decrease the execution time compared to the static worker pool, because it allows idling processing units to enter a deep sleep idle state which again boosts the remaining active workers with the saved energy budget.

5.5.3 Results

Figure 5.12 presents the measured parallelism profile of ODTLES using TBB with static worker pool on MyThOS. As expected, the application operates in alternating phases with different degree of parallelism. The sequential parts require 73.62 percent of the total execution time in this scenario. All available 112 worker threads are only simultaneously active for 0.16 percent of the execution time and the average parallelism is 15.03. From this it can be derived that even in the parallel phases of the application there is not enough parallel work to keep all available workers busy. Additionally, the application will not take

huge advantage from additional processing resources due to the already large proportion of sequential phases. However, at the beginning of each parallel phase, all available workers are activated, trying to support the processing of the parallel workload. Unfortunately, after a series of failed work stealing attempts, some of those workers go back to sleep without doing useful work. For resource management, this means that all unallocated processing units in the system, which may even be in a deep sleep state, are woken up and shortly thereafter put back to sleep, resulting in a waste of energy and thus computing capacity. This behavior is caused by the TBB task scheduler, which wakes up all available workers only for the case of a potentially large amount of parallel work to be generated[22].

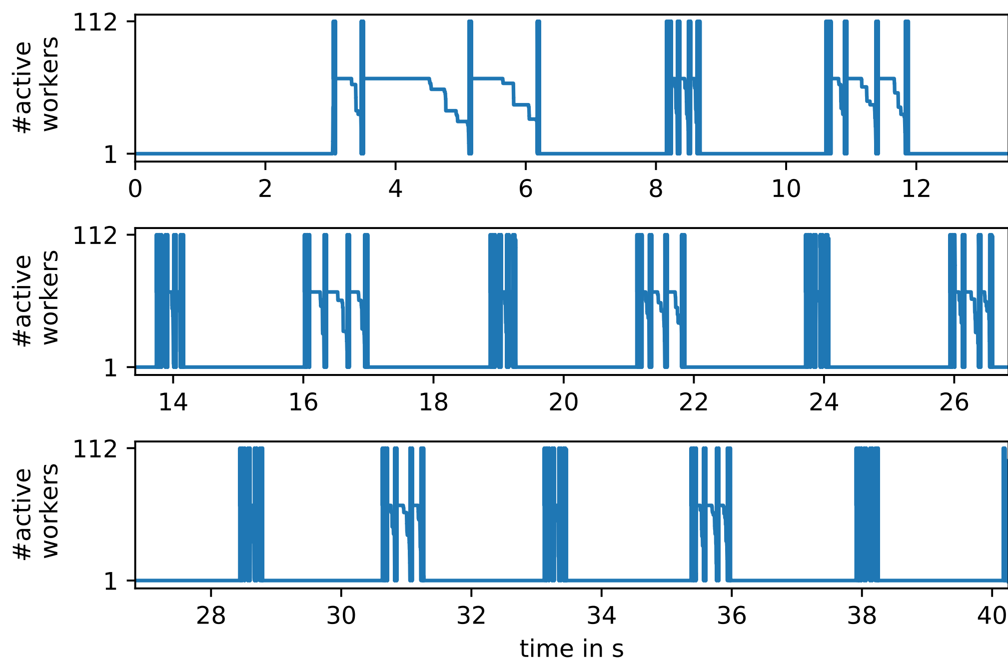


Figure 5.12: Parallelism profile of ODTLES benchmark in terms of TBB worker activity with static worker pool on MyThOS

The scalability in execution time of the ODTLES benchmark depending on the worker thread allocation scheme is shown in figure 5.13. The sequential execution time, where no further processing units are assigned to the application, is 141.8 seconds and identical for both worker allocation schemes. This meets the expectation because no dynamic worker reallocation takes place and all unused processing units are kept in deep sleep by the processing resource management. Using multiple worker threads reduces the execution time in both scenarios. Up to 16 threads, their execution times are nearly identical. From 32 to 112 threads, dynamic worker allocation causes a lower execution time compared to the static worker pool. Both

scenarios experience a knee in the speedup curve, where the execution time increases with more threads. The static worker pool variant has its lowest execution time with 38.35 seconds when using 32 worker threads, which then increases to 40.67 seconds when using all 112 worker threads. The dynamic worker pool reaches its lowest execution time at 64 worker threads with 35.07 seconds, which increases to 35.9 seconds when using all 112 threads. Thus, the dynamic worker pool offers increased scalability. This can be explained by the deeper target sleep state of idling hardware threads during sequential phases, which allows increased frequency boosting of the active threads.

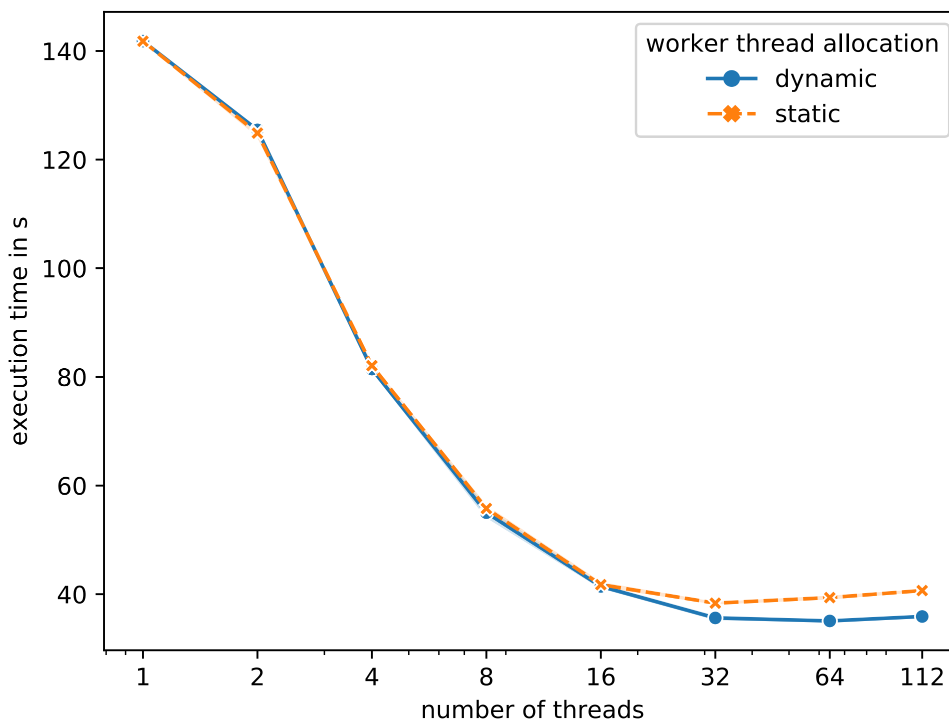


Figure 5.13: Execution time of the ODTLES benchmark using TBB on MyThOS with the different processor allocation mechanisms depending on thread limit and worker allocation approach

Figure 5.14 shows the energy consumption of both processor packages required for a benchmark iteration. As expected, the energy consumed is almost proportional to the execution time. In both scenarios, sequential execution requires 16.1 kilojoule and decreases when using more worker threads. Analogous to the measured execution time, the energy consumption of the worker allocation schemes begins to differ when using more than 16 worker threads. So, the static worker allocation scheme has its minimum at 32 threads with 5.51 kilojoule which again increases to 6.02 kilojoule when using all 112 threads. The energy consumption of the dynamic worker allocation scheme reaches its minimum at 64 threads with 5.12 kilojoule which increases to 5.48 kilojoule when using all 112 threads. Still, using more threads than

the optimal number increases the energy consumption more than the execution time. This meets the expectation because those extra threads require extra energy, but have only little influence on the execution time.

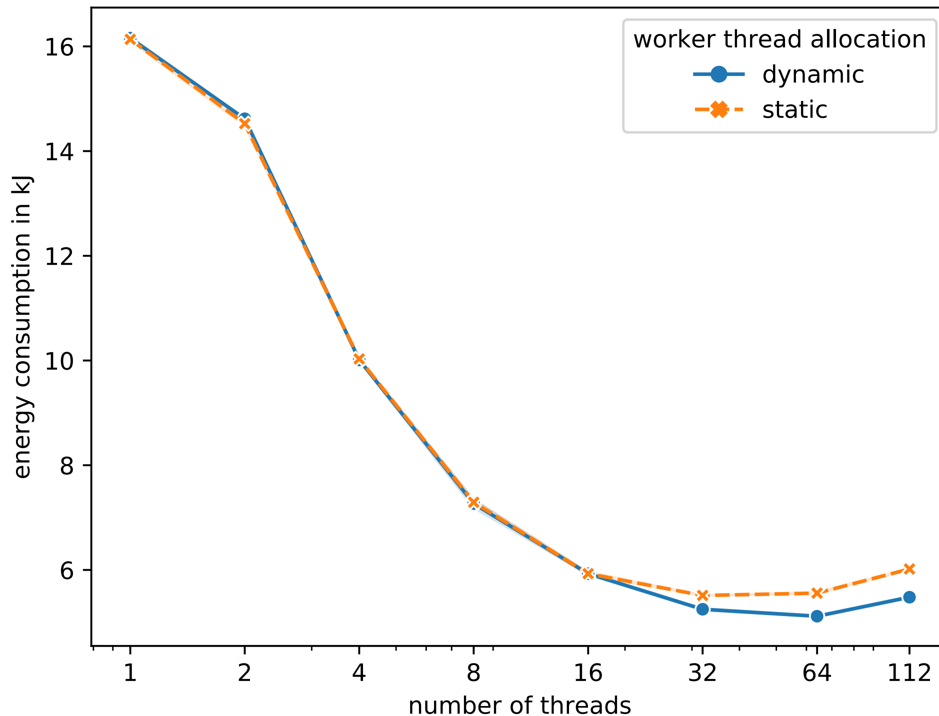


Figure 5.14: The ODTLES benchmark energy consumption of both processor packages using TBB on MyThOS with the different processor allocation mechanisms depending on thread limit and worker allocation approach

5.5.4 Conclusion

This experiment focused on the influence of dynamic worker thread allocation on a real-world application that runs in alternating phases with different resource requirements in the parallelism profile. Therefore, ODTLES[42, 43, 64] was used as a benchmark application. Measurements have proven that this application runs in phases with different parallelism and thus offers potential for dynamic optimization of resource allocation. The experiment has shown that, in comparison to a static worker pool, the dynamic worker allocation reduces the execution time and energy when using more than 16 threads, which equals one eighth of the system's processing resources, and has no disadvantages when using fewer threads. Thus one can conclude that, for the given application, dynamic worker thread allocation offers a suitable approach to allow for dynamic allocation optimization when

using the developed processing resource management. In this way, the runtime system automatically provides information about the application's dynamics in parallelism to the processor management which enables improved dark silicon management or reallocation of resources to other applications. Nevertheless, the success of this mechanism still depends on the cooperative behavior and efficient utilization of processing resources by the user. To detect and conquer such wasteful resource occupancy, the following experiment evaluates the developed dynamic resource redistribution mechanism based on online profiling of the applications.

5.6 Resource Efficiency Control through Online Application Profiling and Resource Revocation

The previous experiment concluded that, for ODTLES[42, 43, 64] with TBB as the benchmark application, dynamic worker allocation in combination with the proposed processing resource allocation mechanism not only does not harm performance, but is actually beneficial for performance and energy efficiency compared to a static worker pool. Furthermore, it provides information about the application's dynamics in parallelism to the processor management which enables improved dark silicon management or reallocation of resources to other applications. However, the success of this mechanism depends on the cooperative behavior and efficient utilization of the processing resources by the user. Therefore, if single applications do not regularly release and reallocate their resources according to their parallelism profiles or do not use the allocated resources efficiently, other applications might suffer from resource deficiency and the overall system efficiency decreases. To detect and conquer such wasteful resource occupancy, this experiment evaluates the dynamic resource redistribution mechanism developed based on online profiling of applications.

5.6.1 Setup

As in the previous experiment, the TBB version of ODTLES[42, 43, 64] is used as a benchmark application. The application is executed once using TBB with a static worker pool as a wasteful and uncooperative variant and is compared to TBB with dynamic worker allocation as a cooperative one.

Online Profiling

As examined in section 3.4, this work proposed determining the application's efficiency using online profiling. Therefore, the efficiency is indirectly measured using the processor's performance monitoring facilities. The actual available performance monitoring events are processor specific. As in the previous experiments, the evaluation is performed on a DELL PowerEdge R740 server system containing two *Intel Xeon Gold 6238R* processors with 28 cores and 56 hardware threads each. These processors are based on the *Cascade Lake* microarchitecture and support Intel's architectural performance monitoring version 4 [18, 20]. Each hyperthread owns three fixed-function counters and four general purpose counters. The fixed-function counters count the number of instructions retired, the number of core cycles while the thread is not in halt state, and the number of TSC reference cycles while the thread is not in halt state. The general purpose counters can be configured to count a specific event from a large selection.

This work focuses on mechanisms rather than heuristics. Thus, the efficiency determination in this experiment serves only as a proof of concept and is not expected to yield optimal results. The number of unhalted reference cycles in relation to the measurement interval in TSC cycles is used to determine the ratio of active cycles during which an allocated hyperthread was not in the halt state due to synchronization using, e.g. a FUTEX. So, the efficiency reduction caused by passive waiting can be quantified. In addition to waiting, interference with shared hardware resources can affect efficiency. An important shared resource in the target system is the memory subsystem that includes the caches. In order to detect efficiency degradation caused by cache contention, the general purpose performance counters are used to observe cache activity. More specifically, this experiment monitors the OFFCORE_REQUESTS_OUSTANDING event with the CYCLES_WITH_DATA_RD mask so that it "counts cycles when offcore outstanding cacheable Core Data Read transactions are present in the super queue. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation) "[79]. The efficiency value is calculated by subtracting the cycles spent on cache transactions from the unhalted reference cycles and divided by the TSC cycles of the interval. The performance monitoring counters are only accessible in kernel mode and by the corresponding hardware thread itself. Hence, allocated threads that are currently in halt state due to passive waiting need to be woken up in order to read their performance counter at the end of each measurement interval. This causes the thread to exit the halt state, which again would distort the efficiency measurement. Thus, the performance counters are configured to count

only unhalted cycles and cache transaction cycles when the corresponding hardware thread is in user mode. A side effect of this decision is that all time spent in kernel mode, which includes system calls as well, is rated inefficient. The measurement interval is set fixed at 100 milliseconds.

Resource Revocation

If processes do not use their allocated processing resources efficiently, the resource management can revoke resources from the processes to increase the efficiency. As described in section 4.2, the processor management uses a PID controller to dynamically determine an upper bound for the partition size of each application based on the measured efficiency. If the number of allocated processing elements exceeds this limit, surplus resources are revoked and can then be reallocated to other processes or put to sleep to conserve energy. The configuration values of the PID controller are empirically determined and set to $K_P = 2.8 * 10^{-1}$, $K_I = 3.92 * 10^{-5}$, and $K_D = 5.6 * 10^{-5}$. They are considered tuning parameters and are not expected to deliver optimal results but to prove the redistribution mechanism. The target efficiency is set to 80 percent. For the static worker pool variant, the revocation handler migrates interrupted worker threads to the main thread and schedules them using time-sharing. Interrupted workers finish their current task and terminate themselves. The revocation handler for the dynamic worker pool enqueues interrupted workers in a global queue. Active workers regularly poll this queue when out of work. If an interrupted worker is found by an active worker, it migrates the interrupted worker to its hardware thread and terminates itself in favor of the interrupted worker.

5.6.2 Expectations

After a sequential initialization phase, the benchmark application allocates additional processing units for parallel execution. While the static worker pool keeps all allocated resources until the end of the program, the dynamic worker pool releases its resources when not used any longer and reallocates them when new parallel work arises. For both types, the static worker pool and the dynamic worker pool, the efficiency of a sequential run, where the application is manually limited to a single processor, is expected to equal one, because no interference with other threads can happen, and it meets the definition of efficiency for a sequential application. When multiple threads are assigned to the application, the efficiency is expected to drop because, as measured in the previous experiment, the speedup of ODTLES

scales sublinearly. The static worker pool is expected to cause lower efficiency compared to the dynamic worker pool because it potentially reduces efficiency by passive waiting in phases of low parallelism. The revocation mechanism is expected to enforce the target efficiency by limiting the maximum number of allocated resources.

5.6.3 Results

This section presents and explains the benchmark results. At first, the measured efficiency profiles without intervention in resource allocation are examined. Subsequently, the influence of dynamic resource revocation on efficiency is investigated.

Online Profiling

Figure 5.15 visualizes the efficiency and thread allocation profile of the ODT benchmark application using TBB with a static worker pool. The application starts with a sequential phase and allocates all available 112 hardware threads at the beginning of the first parallel phase and keeps them until the end of the program. This meets expectations. In theory, the efficiency during the sequential phase is expected to equal one. However, the measured efficiency is lower and even briefly drops below 25 percent. This can be explained by high cache activity for data initialization, which results in an efficiency measurement error. After the static worker pool is allocated, the efficiency drastically decreases, but indicates alternating phases of sequential and parallel processing. During sequential phases the efficiency is below one percent which is caused by passive waiting of inactive workers. In parallel phases, the efficiency increases to approximately only 25 percent due to contention on the shared caches. This also meets expectations. On average, the application occupies 103.08 processing units with an efficiency of eight percent.

The influence of dynamic worker pool allocation on the efficiency profile is shown in figure 5.16. During the sequential initialization phase, it behaves like the variant with a static worker pool. After that, the processing resources are as expected dynamically allocated and released according to the available parallelism. It is noticeable that the profile of allocated threads never reaches the maximum number of 112 threads. This is caused by the fact that those are only average values during the measurement period of 100 ms in which threads are just allocated or released and thus count only partially. A high number of active worker threads decreases efficiency due to contention on the shared caches. On average, the application with

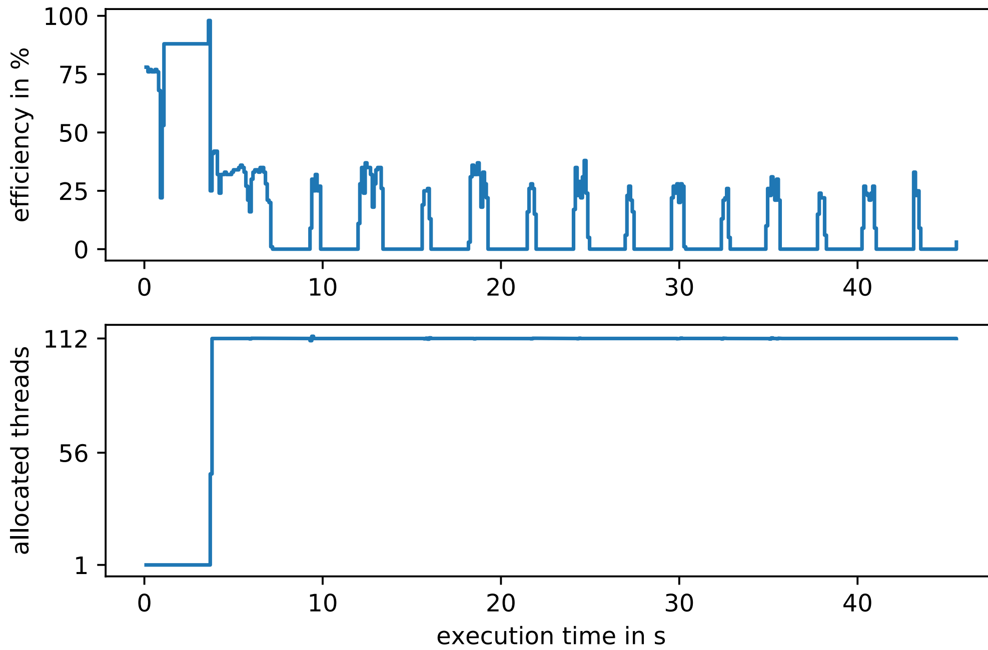


Figure 5.15: Resource allocation and efficiency profile of the ODTLES benchmark using TBB with static worker pool on MyThOS with the processor allocation mechanism

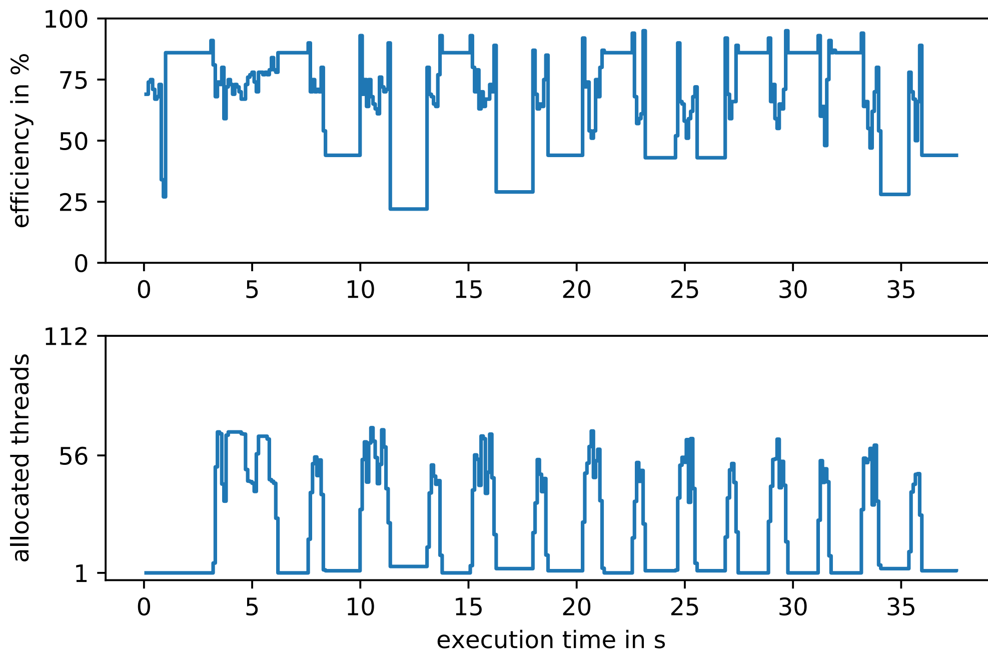


Figure 5.16: Resource allocation and efficiency profile of the ODTLES benchmark using TBB with dynamic worker pool on MyThOS with the processor allocation mechanism

dynamic worker pool occupies 17,61 processing units with an efficiency of 68 percent. Thus, the dynamic worker pool allocation results in an increased efficiency compared to the static worker pool which meets the expetation.

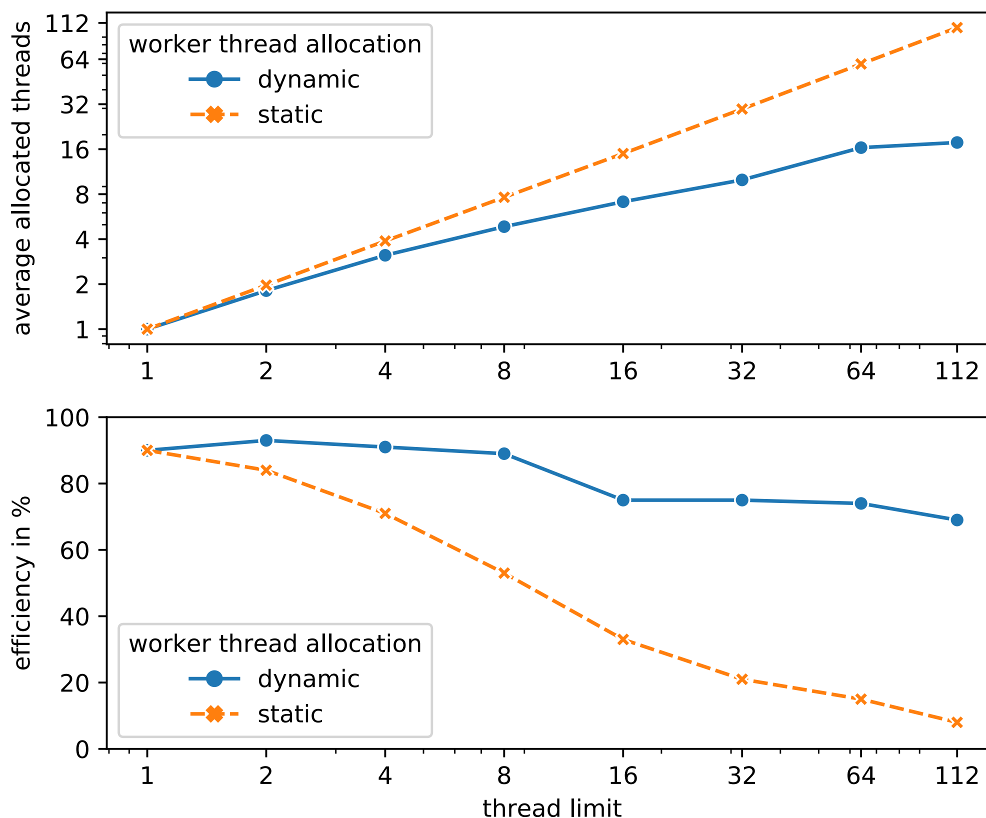


Figure 5.17: Average efficiency and resource allocation of the ODTLES benchmark on MyThOS with the processor allocation mechanism depending on the manual thread limit and worker allocation approach

Figure 5.17 shows the average processing resource allocation and efficiency for ODTLES with both static and dynamic worker pool allocation when scaling the manual thread limit. The static worker pool permanently occupies nearly all available processing elements so that the number of average allocated threads scales linearly with the manual thread limit. The dynamic worker pool, on the other hand, occupies less resources in form of processor time because it releases its worker threads when unused. Against expectations, the measured efficiency when using only a single hardware thread is with 90 percent less than the expected 100 percent of the definition. This can be explained by remote cache accesses, because the data do not fit into local caches, which are interpreted as cache contention and system calls for failing processing resource allocation attempts, which are also rated inefficient. Thus, the implemented efficiency measurement exhibits a certain inaccuracy. As expected, the

efficiency of the static worker pool continuously decreases with a higher number of threads due to more wasted processor cycles of passive waiting in sequential phases. The efficiency of the dynamic worker pool, on the other hand, overall decreases less but actually increases at first from one to two used threads. This can be explained by less contention on the caches due to increased available cache size when using two processing elements and by additional parallelization overhead of the application that is not measured by the profiling and thus rated as useful work.

Dynamic Resource Revocation

The influence of dynamic resource revocation on the efficiency and thread allocation on the ODTLES application with static worker pool allocation is shown in figure 5.18. After a sequential initialization phase, TBB allocates all available processing elements to build a static worker pool. This reduces the current efficiency. Resource profiling automatically detects inefficient resource usage, reduces the thread limit for this application, and revokes allocated resources to enforce this limit. This results in an average of 1.63 allocated processing units with an efficiency increased from 8 to 72 percent. Hence, resource management can automatically detect and counteract inefficient resource usage, which meets expectations. However, the required execution time of the application increases and becomes even greater than that of the sequential variant, but at this point, we are not aiming for minimum execution time of a single application. Also, if there are more processes, the total execution time of all processes could be reduced. The increase in execution time in this case is caused by the overhead for migrating, finishing, and terminating revoked worker threads by the main thread. In addition, TBB reallocates new worker threads at the beginning of each parallel program section according to the current resource limit, which reduces efficiency because they could be revoked again. This is an issue of the current TBB implementation when using a static worker pool and should motivate the programmer to directly develop resource-efficient code.

Figure 5.19 shows the impact of resource revocation on the benchmark application with TBB and dynamic worker allocation. The high number of worker threads created in the first parallel program section reduces efficiency due to cache contention. This is detected by the resource profiling and the thread limit becomes steadily reduced to increase efficiency. So, the number of dynamically allocated worker threads per parallel section decreases over time. The average number of allocated processing elements is 13.67 and the average efficiency increased from 68 to 76 percent. The graph of the dynamic thread limit indicates a slow adaptation

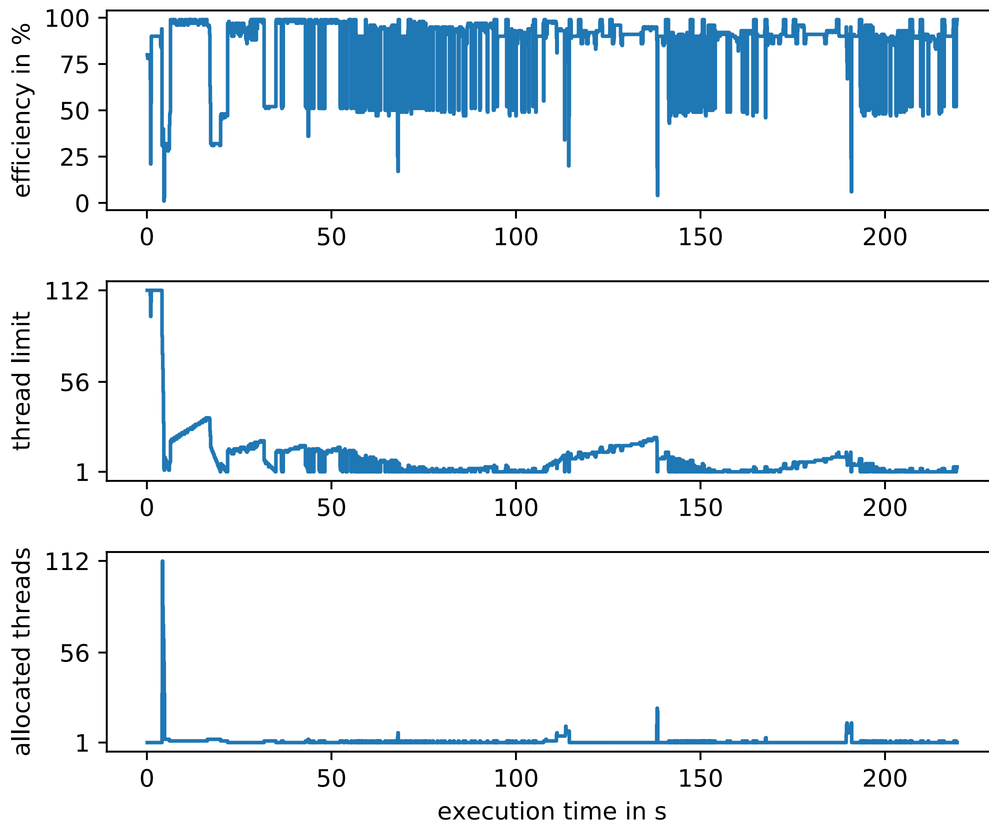


Figure 5.18: Efficiency, dynamic thread limit, and resource allocation profile of the ODTLES benchmark using TBB with static worker pool on MyThOS with the processor allocation mechanism and dynamic resource revocation

to the suitable partition size for this application. This can possibly be improved by further tuning the configuration values of the PID controller or using other heuristics.

5.6.4 Conclusion

The success of energy-efficient spatial processor partitioning depends on the cooperative behavior and efficient utilization of the processing resources by the user. Therefore, if single applications do not regularly release and reallocate their resources according to their parallelism profiles or do not use the allocated resources efficiently, other applications might suffer from resource deficiency and the overall system efficiency decreases. The experiment has shown that the developed resource management mechanism is able to automatically detect and counteract wasteful resource occupancy. In addition, it informs user applications about revoked resources, giving them the opportunity to handle those revocations without

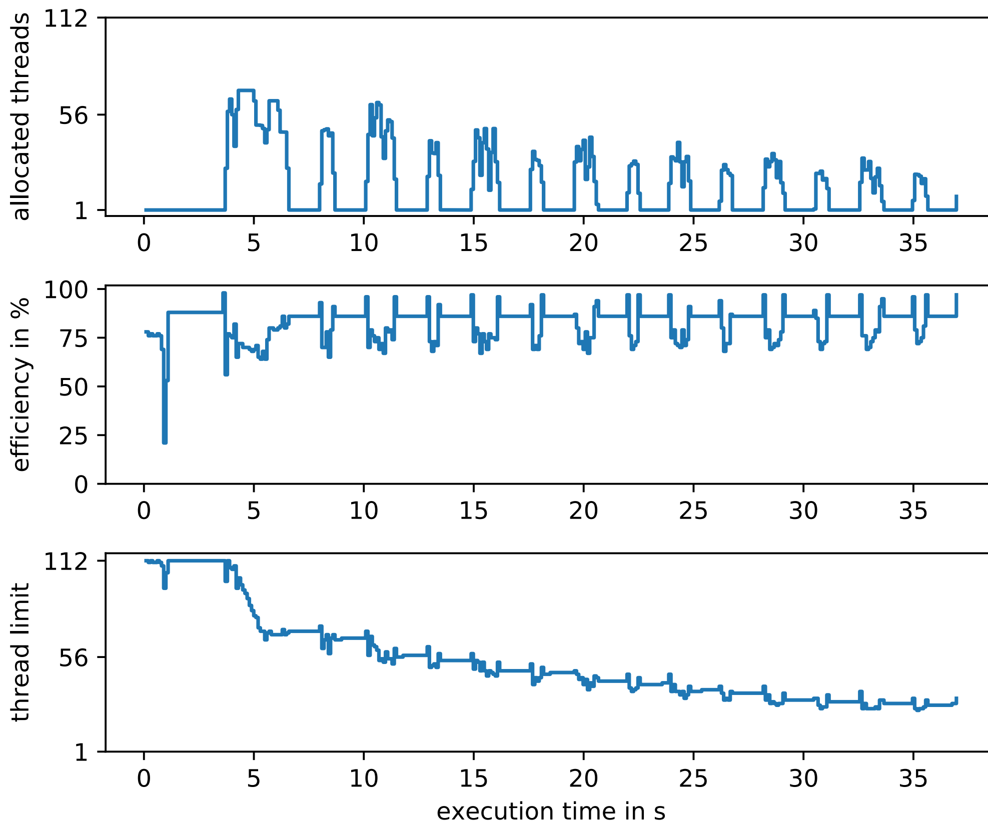


Figure 5.19: Efficiency, dynamic thread limit, and resource allocation profile of the ODTLES benchmark using TBB with dynamic worker pool on MyThOS with the processor allocation mechanism and dynamic resource revocation

crashing the process. The measured efficiency underlays a certain inaccuracy, but is still suitable to indicate inefficiency. Also, the strategy of the efficiency controller to determine a suitable maximum partition size, based on measured efficiency, causes a certain jitter and, in some cases, slow adaptation of the control value. This can potentially be improved by further tuning the configuration values of the PID controller or by using other heuristics. While this experiment evaluated the efficiency control for single applications, the following experiment examines the influence of the resource management when running multiple processes simultaneously, which compete for processing resources.

5.7 Optimizing Energy Efficiency using Dynamic Resource Redistribution

The previous experiment evaluated the detection and counteraction of inefficient resource occupancy of applications running alone on the system, without interference with other applications. The goal of the developed dynamic resource allocation and redistribution mechanisms is to increase the overall energy efficiency of the system and the performance for all applications. Therefore, this experiment examines the impact of these mechanisms when multiple processes are executed simultaneously.

5.7.1 Setup

ODTLES[42, 43, 64] with TBB and dynamic worker allocation is used as a benchmark application with the same configuration as in the previous experiment. The efficiency determination and configuration values of the PID controller for efficiency control remain unchanged, as well. Two processes with the ODTLES benchmark application are created within an initial process and the execution time and energy consumption of the processor packages is measured from the start until both processes finished, which is repeated for consecutive and simultaneous process execution. Both processes are identically configured, but own an individual address space, capability space, and use memory from separate NUMA nodes for their heaps to minimize interference.

As in the previous experiments, the processor pool balancing thresholds are configured as follows. When the number of free resource in a thread team falls below one hardware thread and the applications efficiency meets the target efficiency, more resources are requested from the central processor allocator. When a thread team owns more than four free hardware threads, it returns, if possible, a whole core including its two associated threads back to the central processor allocator, which makes these threads available for allocation by the team of the other process. Thus, a thread team strives to always have at least one free hardware thread in low latency sleep available and no more than four hardware threads per team are prevented to enter a deep sleep state in the central processor allocator. The target idle sleep states are configured to *deep sleep* for all threads owned by the central processor allocator and *halt* as the target sleep state for all idle threads assigned to a team. So, while hardware threads owned by the central processor allocator are not currently assigned to a specific process and considered cold in terms of wakeup latency and caches, hardware threads owned

by the team provide a higher hotness. They have a lower wakup latency and their local caches potentially contain application data from previous assignments which predestines them to serve high frequent allocation requests of their corresponding process.

5.7.2 Expectations

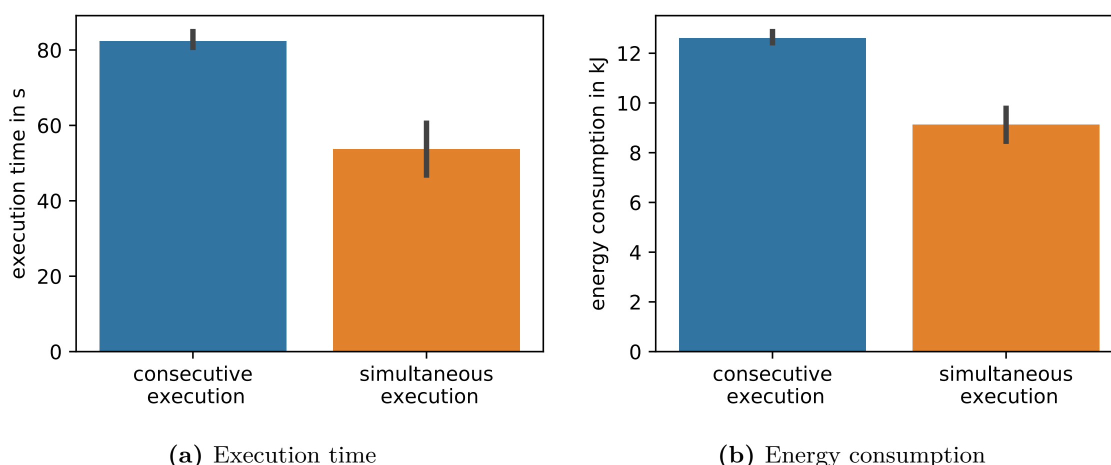
The simultaneous execution of both processes is expected to result in a lower overall execution time and energy consumption then the consecutive execution of both processes, because the developed resource allocation and efficiency control mechanism will ensure efficient allocation and redistribution of processing resources. Hence, greedy and inefficient resource allocation will be prevented and each process has a change to get a suitable partition.

5.7.3 Results

Figure 5.20 presents the overall execution time and energy consumption of two processes running ODTLES using TBB with a dynamic worker pool on MyThOS using the proposed processing resource management and dynamic efficiency control. When running both processes simultaneously, the median execution time was reduced by 36,52 percent from 81.38 seconds to 51.66 seconds and the median energy consumption of the processor packages by 28,54 percent from 12.51 kilojoule to 8.94 kilojoule, compared to consecutive execution. This meets the expectation. The scattering of the simultaneous execution time and energy consumption is caused by the efficiency control strategy, which only limits the partition size of the local process, based on the measured efficiency, but does not take other processes into account. This could potentially be improved by a global balancing strategy.

5.7.4 Conclusion

The developed dynamic resource allocation and redistribution mechanisms are able to increase overall energy efficiency and application progress when running multiple processes simultaneously. The process-local efficiency control strategy could possibly be improved by combining it with a global redistribution strategy to allocate resources more evenly among the processes with respect to their individual efficiency. However, this work aims for mechanism rather than strategies, which thus could be optimized in a future work. A summary about the evaluation results in provided in the following.



(a) Execution time

(b) Energy consumption

Figure 5.20: Execution time and energy consumption of the simultaneous and consecutive execution of two processes running ODTLES using TBB with dynamic worker pool on MyThOS with the processor allocation mechanism and dynamic resource revocation

5.8 Summary

The processor allocation and redistribution mechanism developed in this work strives to increase the system global energy efficiency on multi- and manycore systems. It follows the one-thread-per-core execution model to avoid the overhead of regular context switches and systematically puts unused cores, considering the actual hardware topology, into a specific sleep state to reduce energy consumption and accelerate active cores.

When a processor is currently unused, deeper C-states bring higher energy savings but at the cost of increasing wakeup latency when needed again. Experiments have proven those expectations and shown that using *CC3* instead of *CC1* sleep state offers only a little benefit in the form of power savings but leads to a significant increase in wakeup latency. Therefore, one can conclude that putting unused processor cores into a sleep state is a worthy way to dynamically decrease idle power consumption, but suffers from wakeup latency. To compensate this and the power loss of frequent state transitions, a sophisticated OS-level processor management is required.

Existing parallel runtime systems try to reduce the overhead of frequent thread allocation by maintaining static thread pools. This behavior hides the dynamics of the application parallelism from the OS and, thus, complicates the resource and power management of the OS. In addition, when using spatial processor partition with exclusive allocation, it prevents dynamic reallocation which results in underutilization. The developed processor allocation

mechanism that is implemented in MyThOS reduces the allocation latency of a software thread by more than 80 percent compared to Linux. This is not only directly beneficial for performance and energy efficiency but also encourages applications and parallel runtime systems to dynamically allocate and release processing resources.

The developed resource management in MyThOS limits the number of allocated software threads to the number of physically available hardware threads. This requires the application to act resource-aware and thus being able to handle denied resource allocation requests. In combination with the reduced thread allocation latency, this results in more than seven times faster execution of the parallel mandelbrot set rendering application than Linux, which does not limit the number of parallel software threads and, therefore, causes massive overhead for thread allocation and rescheduling.

The experiments have shown that it is more energy efficient to process the mandelbrot application using a high number of active processor cores and, thus, finish execution earlier than using only a single hardware thread and putting others to sleep. This trend is still valid when considering a fixed time interval instead of a fixed application size and putting all cores into a deep sleep state after termination. From the results, one can derive that a proper use of processor idle sleep states has a major impact on the energy efficiency as well as the performance, especially when dynamic performance boosting is activated and not all cores are permanently in use.

Existing parallel runtime systems facilitate the development of parallel applications, because they implement the basic functionality of the parallel execution model. This work proposes to shift the dynamic processing resource handling into the parallel runtime system as well, to disburden the application programmer from the manual implementation. This strives to increase the productivity by reducing the code complexity and thus susceptibility to errors. It has been proven that using TBB as a task-parallel runtime system leads to a reduction in execution time from 3.2 percent up to 58.8 percent and a decrease of energy consumption from 3.8 percent up to 45.4 percent depending on the number of assigned processing units compared to the manual thread allocation in the mandelbrot benchmark. From this one can conclude that the usage of a task-parallel runtime system is not only not harmful but beneficial for energy efficiency and thus represents a suitable basement for automatic processing resource handling.

Measurements have proven that ODTLES, taken as an example for a real world scientific application, runs in phases with different parallelism and thus offers potential for dynamic optimization of resource allocation. The experiments have shown that, in comparison

to a static worker pool, the dynamic worker allocation reduces the execution time and energy consumption when using more than 16 threads and has no disadvantages when using fewer threads. Thus one can conclude that, for this application, dynamic worker thread allocation offers a suitable approach to allow for dynamic allocation optimization when using the developed processing resource management. In this way, the runtime system automatically provides information about the application's dynamics in parallelism to the processor management which enables improved dark silicon management or reallocation of resources to other applications.

The success of energy-efficient spatial processor partitioning depends on the cooperative behavior and efficient utilization of the processing resources by the user. Therefore, if single applications do not regularly release and reallocate their resources according to their parallelism profiles or do not use the allocated resources efficiently, other applications might suffer from resource deficiency and the overall system efficiency decreases. The developed efficiency control mechanism has been proven to be able to automatically detect and counteract wasteful resource occupancy. In this case, it informs user applications about revoked resources, giving them the opportunity to handle those revocations without crashing the process. Additionally, it has been shown that the developed dynamic resource allocation and redistribution mechanisms are also able to increase overall energy efficiency and application progress when running multiple processes simultaneously.

CHAPTER 6

Conclusion

Multi- and manycore processors are the microprocessor industry's response to the constantly growing demand for computational power and energy constraints of today's CMOS technology. Instead of increases in core sophistication and clock rate, more and more, generally simpler processor cores are integrated into a single chip to achieve a higher overall peak performance without increasing power consumption and complexity of each single core. In this manner, the energy efficiency in form of computation per watt can be improved by scaling the energy consumption linearly with the number of cores instead of exponentially with the frequency and voltage. Thereby, the former trend of integrating a huge amount of simple in-order cores in a manycore processor went to a smaller but increasing number of more complex and powerful out-of-order cores. In the near future, processors will contain hundreds to thousands of cores.

Although manycore processors provide a high peak performance, they suffer from thermal constraints causing dark silicon. Hence, not all circuits of a processor can be operated permanently at full frequency. Consequently, the maximum performance of those processors is limited both by the amount of energy available and by the dissipation of power loss in the form of thermal energy.

Traditional OSs still rely on time sharing to create the illusion of a dedicated machine for the user and allow to run more processes pseudo-simultaneously than physical processor cores are available. To ensure fair execution between all processes, the operating system has to perform periodic context switches between the software threads on the processor cores. This introduces a significant overhead to the actual productive work, wastes energy and thus reduces computational performance. Application developers adapted to the time-sharing abstraction and to the high costs of software thread creation on state-of-the-art OSs by maintaining static thread pools to avoid the overhead of regular thread allocations. Unfortunately, this behavior hides the application's dynamics from the processor resource management and thus complicates dark silicon management in the OS.

This work investigated mechanisms for energy-efficient processor allocation and redistribution. In order to avoid the costs of frequent context switches, time sharing is replaced by spatial partitioning which grants processes exclusive access to allocated processor cores, but requires resource-aware behavior. Processing resources are hierarchically managed to scale with the increasing number of cores in future manycore systems. Currently unused processor cores are put to sleep in order to save energy and boost active cores. To balance the trade-off between maximum energy savings and minimal allocation response time, idling processor cores are organized in multiple pools with different sleep states. Hence, a fast thread allocation has been achieved which motivates applications for dynamic thread allocation and benefits performance as well as energy efficiency.

Spatial partitioning and the one-thread-per-core execution model limit the number of allocated software threads to the number of physically available hardware threads. This prevents over-subscription but requires dynamic partitioning of the system. The partitioning strategy respects the processor topology, because it is coupled to the memory hierarchy, communication distances, and power management domains. The efficiency control and resource revocation mechanisms detect and prevent wasteful and inefficient resource occupation from poorly optimized or malicious processes. In this way, the system's global efficiency can be optimized instead of locally seeking for the knee in the speedup curve of each individual application. The dynamic processing resource allocation and revocation handling has been integrated into a task parallel runtime system to disburden the application programmer from the manual implementation. This increased productivity by reducing code complexity and thus susceptibility to errors.

Although this work made several contributions in the field of energy-efficient resource management, a few concerns remain. This work focused on mechanisms, but there are multiple strategies employed to prove the concepts. This include placement of partitions in the processor topology tree, balancing of resources between the processor pools, selection of performance monitoring events for efficiency determination, and partition size limitation based on the measured efficiency. Those strategies employed for this work are considered non-optimal and replaceable. However, further research could benefit performance, energy-efficiency, and reduce management overhead on the application side caused by fluctuating partition limitations. Machine learning might be a suitable method to improve a subset of those strategies.

The proposed resource management does not assume any static knowledge or an preceding offline profiling of the application. Additionally, information from the application, respectively,

the runtime system, is not trusted because it might be inaccurate and the information of different processes might not be comparable. Consequently, application profiling was performed online by the OS only using standard PMUs. However, accepting hints from the user about future resource requirements could improve decision making. So, e.g. the placement of software threads in the processor topology could be optimized when getting information about the programs properties. This includes whether it is memory or computation bound, its communication patterns with other threads, or future expansion and reduction in parallel work. Also, this could allow preheating of cores in the mean of activating cores and loading the working set data into the caches before the actual work becomes available.

As described in section 4.3, TBB was adapted to dynamically allocate workers at the beginning of a parallel section and release them when out of work. Unfortunately, the client side of TBB is programmed to always immediately request as many workers as hardware threads available in the system, even if only a single task is created. While this behavior is beneficial when potentially entering a program phase with massive parallelism and thus waking up all workers in a static worker pool, it causes wasteful allocation and direct termination of worker threads when using a dynamic worker pool due to a shortage of parallel work. Hence, an additional modification of TBB, so that it only requests as many workers as actual work is available, which is known as adaptive work stealing, could further reduce overhead and improve efficiency.

APPENDIX A

Energy Dissipation in CMOS Processors

The total power consumption in CMOS technology arises from several sources. Thereby, it can be divided into static and dynamic components[61]:

$$P_{total} = P_{static} + P_{dynamic}$$

A.1 Static Power Consumption

Static power consumption occurs when all inputs of a circuit are held in some constant and valid logic level and no charging or discharging is required. For clear illustration, figure A.1 shows the circuit of a CMOS inverter that is typically included in all input and output stages of CMOS devices. An inverter contains a P-Channel Metal Oxid Semiconductor (PMOS) and a N-Channel Metal Oxid Semiconductor (NMOS) that are connected at the gate and the drain terminals. The supply voltage is connected to the source terminal of the PMOS and ground to the source terminal of the NMOS transistor. Both gate terminals are connected to the input voltage and the output voltage is connected to the drain terminals. If the input voltage is at logic 1 then the NMOS transistor is ON (low resistance) and the PMOS transistor is OFF (high resistance). Hence, the output voltage equals ground which is logic 0. When the input level is logic 0, the states of both transistors switch and output is 1.

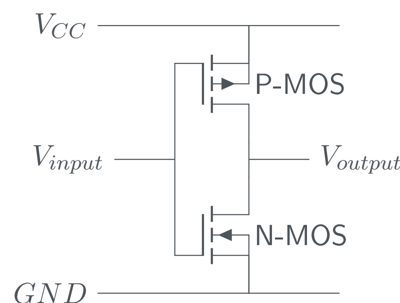


Figure A.1: CMOS inverter circuit

Due to the different channels, the PMOS and NMOS transistors have an opposite switching state at both possible logical input values. So, there is always one transistor in high resistance state preventing a short circuit between the supply voltage and ground. This property makes CMOS energy efficient compared to previous Transistor–Transistor Logic (TTL) circuits. However, due to a small leakage current, the static power consumption does not equal zero. The leakage current is caused by parasitic diodes between the N-well and the P-substrate of the CMOS inverter. The amount of leakage current adds up from all CMOS devices and their parasitic diodes and depends on the voltage at each parasitic diode and the temperature. Thus, the static power consumption equals the total leakage current multiplied by the supply voltage:

$$P_{static} = I_{leakage} * V_{cc}$$

A.2 Dynamic Power Consumption

In contrast to static power consumption, dynamic power consumption only occurs when switching between logical states in the circuit and depends on the switching frequency. Hence, when switching at high frequency, dynamic power consumption contributes a high portion of the overall power consumption. Charging and discharging of capacitive load at the output increases the dynamic power consumption even further. The dynamic power consumption of a CMOS circuit consists of *transient power consumption* and *capacitive-load power consumption*:

$$P_{dynamic} = P_{transient} + P_{capacitive_load}$$

A.2.1 Transient Power Consumption

Transient power consumption is caused by the current that flows when switching transistors from one logic state to another. Thereby, the internal transistors need to be charged or discharged, moving charges in the parasitic capacitors on the CMOS gates which creates a switching current. For dynamic power approximation, the power-dissipation capacitance (C_{pd}) is specified as a measure of the equivalent capacitance of a CMOS circuit with several gates. Additionally, in logic transition, NMOS and PMOS are both in a conductive state for a short period of time which causes a short circuit between the supply voltage and ground. Transient power consumption can be approximated by multiplying the dynamic

power-consumption capacitance (C_{pd}), the input frequency with which the inputs of the CMOS devices are switched, and the supply voltage squared:

$$P_{transient} = C_{pd} * f_I * V_{cc}^2$$

A.2.2 Capacitive-Load Power Consumption

The second part of dynamic power consumption is caused by the current that charges or discharges capacitive-loads at the output of a CMOS device. Therefore, the capacitive-load power consumption can be calculated by multiplying the load-capacitance (C_L), the frequency (f_O) at which the output value is changed, and the supply voltage (V_{cc}) squared. This has to be calculated and summed up for each individual bit in the CMOS circuit with respect to the individual frequency of the output value switches. Hence, the equation for the capacitive-load power consumption is not accurate but provides an idea about the involved factors:

$$P_{capacitive_load} = C_L * f_O * V_{cc}^2$$

APPENDIX B

CPU Instructions for Entering Idle States

Idle power saving states are used to shut down parts of a processor when not used. Entering those idle states can either be done using the ACPI interface or using a set of hardware specific instructions. Those instructions and their effect on the processor's sleep state are examined in in the following. Mind that they only directly affect the state of the hardware thread they are executed on and the core level and package level C-states are only influenced when all hardware threads of a core, respectively all cores in a package enter a sleep state.

B.1 HLT (Halt)

The *HLT* instruction [18] suspends the execution of the logical processor (enter the *C1-state*) until an interrupt is received. This might be a hardware device interrupt or an Inter-Processor Interrupt (IPI). If a logical processor is halted, other active logical processors within the same package keep full access to shared resources. The remaining logical processors might experience greater efficiency when accessing previously shared resources due to less contention. When a halted logical processor resumes execution, previously shared resources get shared among all active logical processors again. When using hyper-threading, long idle periods or spin-wait loops of one thread while the other hyper-thread is doing useful work should be avoided and the thread explicitly be halted because idling loops consume a significant amount of the processing resources that otherwise would be used be the other thread.

B.2 MONITOR/MWAIT

The *MWAIT* instruction [18, 19] is used in combination with the *MONITOR* instruction to suspend the calling processor's execution until a store to a specific linear address range is performed by another thread. The *MONITOR* instruction sets up the address monitoring hardware to observe a specific memory address range that must be mapped in write-back

caching mode. *MWAIT* sets the processor in an optimized state until a write to the monitored memory area occurs. It has the same effect on the architectural state of the system as the *NOP* instruction. Events like interrupts, TLB invalidations and others can cause a processor to wake up from *MWAIT*, too. Additionally, it expects a hint for which processor-specific (but not Advanced Programmable Interrupt Controller (APIC)-specific) C-state to be entered as an argument. However, internal conditions may cause the processor to ignore the hint and enter a different optimized state. This pair of instructions is only available in kernel mode.

B.3 Intel's UMONITOR/UMWAIT

The *UMONITOR* and *UMWAIT* instructions [18, 19] are the user mode versions of *MONITOR* and *MWAIT* and can be executed on any privilege level. They are only supported on Intel hardware and introduced with *Tremont* microarchiteture. *UMWAIT*, in contrast to *MWAIT*, allows to set up a timer which, when expired, wakes the processor if no store to the observed memory area occurred beforehand.

B.4 AMD's MONIORX/MWAITX

The *MONITORX* and *MWAITX* instructions [57] are the user mode versions of *MONITOR* and *MWAIT* and can be executed on any privilege level. Therefore, they equal Intel's *UMONITOR* and *UMWAIT* instructions. *MWAITX* allows for setting up a timeout as well.

B.5 PAUSE

The *PAUSE* instruction [18] does not instruct the processor to enter a C-state, but improves power efficiency of CPUs with hyper-threading when placed in spin-wait loops.It provides a hint to the CPU that the current hardware thread is executing a spin-wait loop which then prevents the thread from unnecessary excessive execution resource and power consumption while keeping resources available for other threads. This also avoids high energy overhead and wakeup latency when entering C-states using *MONTIOR* or *MWAIT* only for short periods.

B.6 TPAUSE

The *TPAUSE* [18, 19] instruction causes a processor to enter a implementation-dependent optimized state until the TSC reaches or exceeds a certain input value. When calling, the user chooses from two optimized states by the input value. The performance state brings a low wakeup latency for the price of high power consumption, whereas the low power state improves performance for other threads on the same core but requires a longer time for waking up. They are called *C0.1* and *C0.2*. While in implementation-dependent sleep state, the processors can also be woken up by incoming interrupts. This instruction is only available on Intel hardware and was introduced with *Tremont* microarchitecture.

B.7 Conclusion

Modern CPUs offer software interfaces to control the power consumption. *P-states* are operational performance states that allow to control the power consumption while the processor is actively executing code. They are implemented using dynamic voltage and frequency scaling and can either be controlled using ACPI or by activating autonomous Hardware-Controlled Performance States (HWP) with performance hints from the OS, when supported. *C-states* control the idle power consumption when the processor is not execution anything by power gating currently unused subsystems. Core level C-states can directly be controlled using ACPI or platform-specific instructions like *MONITOR/MWAIT*, *HALT*, or *TPAUSE* while package level C-States are automatically derived from the lowest idle state of all cores a package contains. Higher C-states offer a significant power saving potential but bring higher wakeup latency and energy costs for state transition.

APPENDIX C

Acronyms

ACPI Advanced Configuration and Power Interface

API Application Programming Interface

APIC Advanced Programmable Interrupt Controller

CCD Core Complex Die

CCX Core Complex

CFS Completely Fair Scheduler

CGMT Coarse-grain Multithreading

CMOS Complementary Metal Oxid Semiconductor

CPU Central Processing Unit

DRAM Dynamic Random-Access-Memory

FGMT Fine-grain Multithreading

FIFO First-In, First-Out

FPU Floating Point Unit

FUTEX Fast Userspace Mutex

GMI Global Memory Interconnect

HPC High Performance Computing

HWP Hardware-Controlled Performance States

IDA Intel Dynamic Acceleration

IOD I/O Die

I/O Input and Output

IoT Internet of Things

IPC Inter-Process Communication

IPI Inter-Processor Interrupt

iRTSS Invasive Run-Time Support System

LLC Last-Level Cache

LRU Least Recently Used

MPI Message Passing Interface

MPSoC Multi-Processor Systems-on-Chip

MyThOS Many Threads Operating System

NMOS N-Channel Metal Oxid Semiconductor

NoC Network on a Chip

NRU Not Recently Used

NUMA Non-Uniform Memory Access

NVRAM Non-volatile Random-Access-Memory

ODTLES One-Dimensional Turbulence Large Eddy Simulation

ODT One-Dimensional Turbulence

OpenMP Open Multi-Processing

OS Operating System

OSPM Operating System-directed Configuration and Power Management

PCB Process Control Block

PCID Process-Context Identifier

PID controller Proportional–Integral–Derivative controller

PLL Phase-Locked Loop

PMOS P-Channel Metal Oxid Semiconductor

PMU Performance Monitoring Unit

POSIX Portable Operating System Interface

RAPL Running Average Power Limit

RISC Reduced Instruction Set Computer

SMT Simultaneous Multithreading

SPMD Single Program Multiple Data

SRAM Static Random-Access Memory

TBB Intel Threading Building Blocks

TCPA Tightly-Coupled Processor Array

TDP Thermal Design Power

TLB Translation Lookaside Buffer

TLS Thread-Local Storage

TSC Time Stamp Counter

TTL Transistor–Transistor Logic

UMA Uniform Memory Access

XLES Extended Large Eddy Simulation

List of Figures

2.1 Hardware multithreading types 9
2.2 Example of processor tiling architecture with four tiles containing four cores and a local cache each. All tiles, memory, and IO devices are connected via NoC 10
2.3 Examples of Network on a Chip topologies 11
2.4 Application speedup models 18
2.5 Relationship between execution time, speedup, efficiency, and speedup efficiency 19
2.6 Types of processor multiplexing 21
2.7 Parallelism profile of a parallel program 27
2.8 The life cycle of a resource-aware application in OctoPOS[85] 33

3.1 Structure of the processor topology tree 46
3.2 Processor pools 49
3.3 Resource management hardware thread state diagram 50
3.4 The internal structure of a processor pool 50
3.5 Procedure of resuming a suspended worker thread by rebinding its saved software thread context to a processing unit, taken from [41] 66

4.1 Processor allocation architecture overview 72
4.2 Comparison of pessimistic and optimistic context and processor allocation procedures 75
4.3 Control loop to limit the partition size of an application based on the measured efficiency 77

5.1 Processor topology of the evaluation system (DELL PowerEdge R740 Rack-Server containing two Intel Xeon Gold 6238R[20] processors), generated using hwloc[11]. 82
5.2 Wakeup latency depending on core level idle state and substate using *MONITOR/MWAIT* 85
5.3 Processor package power consumption depending on idle state and substate. Measured using RAPL. 87
5.4 Runtime overhead of creating, creating and starting, and creating and joining a posix thread using MyThOS with the developed processor allocation mechanism compared to Linux 91

5.5 Mandelbrot set rendering 92
5.6 Mandelbrot set benchmark execution time comparison Linux versus MyThOS with the proposed processor allocation mechanism in dependency of activated IDA 96
5.7 Execution time of mandelbrot set benchmark on MyThOS with the processor allocation mechanism depending on thread limit and target sleep modes with active IDA 98
5.8 Energy consumption of both processor packages for Mandelbrot set benchmark on MyThOS with the processor allocation mechanism depending on the thread limit and target sleep modes with activated IDA 100
5.9 Mandelbrot set benchmark energy consumption of both processor packages on MyThOS with the processor allocation mechanism depending on the thread limit and target sleep modes scaled to fixed execution time and with IDA activated 101
5.10 Execution time of the mandelbrot set benchmark on MyThOS with the processor allocation mechanism depending on thread limit and parallelization approach 106
5.11 Mandelbrot set benchmark energy consumption of both processor packages on MyThOS with the processor allocation mechanism depending on the thread limit and parallelization approach 107
5.12 Parallelism profile of ODTLES benchmark in terms of TBB worker activity with static worker pool on MyThOS 112
5.13 Execution time of the ODTLES benchmark using TBB on MyThOS with the different processor allocation mechanisms depending on thread limit and worker allocation approach 113
5.14 The ODTLES benchmark energy consumption of both processor packages using TBB on MyThOS with the different processor allocation mechanisms depending on thread limit and worker allocation approach 114
5.15 Resource allocation and efficiency profile of the ODTLES benchmark using TBB with static worker pool on MyThOS with the processor allocation mechanism 119
5.16 Resource allocation and efficiency profile of the ODTLES benchmark using TBB with dynamic worker pool on MyThOS with the processor allocation mechanism 119

5.17 Average efficiency and resource allocation of the ODTLES benchmark on MyThOS with the processor allocation mechanism depending on the manual thread limit and worker allocation approach 120
5.18 Efficiency, dynamic thread limit, and resource allocation profile of the ODTLES benchmark using TBB with static worker pool on MyThOS with the processor allocation mechanism and dynamic resource revocation 122
5.19 Efficiency, dynamic thread limit, and resource allocation profile of the ODTLES benchmark using TBB with dynamic worker pool on MyThOS with the processor allocation mechanism and dynamic resource revocation 123
5.20 Execution time and energy consumption of the simultaneous and consecutive execution of two processes running ODTLES using TBB with dynamic worker pool on MyThOS with the processor allocation mechanism and dynamic resource revocation 126

A.1 CMOS inverter circuit 133

List of Tables

2.1 Examples for multi- and manycore processors[20, 21, 56, 58, 62, 69, 102] . . . 8
2.2 Parallel job types in terms of processor allocation[36] 24
2.3 Comparable concepts of MyThOS in sel4, Nova, and Unix[71] 34

5.1 Expected supported C-States, exit latency and target residency time for *Skylake X* and *Cascade Lake* microarchitecture based processors for the *MWAIT* instruction in the Linux Kernel[104] 83

Bibliography

[1] K. Agrawal et al. “Adaptive work-stealing with parallelism feedback.” In: *ACM Transactions on Computer Systems (TOCS)* 26.3 (2008), pp. 1–32.

[2] B. Alam, M. N. Doja, and R. Biswas. “Finding Time Quantum of Round Robin CPU Scheduling Algorithm Using Fuzzy Logic.” In: *2008 International Conference on Computer and Electrical Engineering*. 2008, pp. 795–798. DOI: `10.1109/ICCEE.2008.89`.

[3] G. M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities.” In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[4] N. Asmussen et al. “M3: A hardware/operating-system co-design to tame heterogeneous manycores.” In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 2016, pp. 189–203.

[5] A. Barbalace, B. Ravindran, and D. Katz. “Popcorn: a replicated-kernel OS based on Linux.” In: *Proceedings of the Linux Symposium, Ottawa, Canada*. 2014.

[6] F. Bartz. “Dynamische Partitionierung von Mehrkernsystemen.” MA thesis. BTU Cottbus–Senftenberg, 2020.

[7] A. Baumann et al. “The multikernel: a new OS architecture for scalable multicore systems.” In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 29–44.

[8] S. Bennett. “Development of the PID controller.” In: *IEEE Control Systems Magazine* 13.6 (1993), pp. 58–62. DOI: `10.1109/37.248006`.

[9] R. D. Blumofe and C. E. Leiserson. “Scheduling multithreaded computations by work stealing.” In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[10] S. Boyd-Wickizer et al. “An Analysis of Linux Scalability to Many Cores.” In: *OSDI*. Vol. 10. 13. 2010, pp. 86–93.

[11] F. Broquedis et al. “hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications.” In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 2010, pp. 180–186. DOI: `10.1109/PDP.2010.67`.

[12] J. R. Bulpin and I. Pratt. “Hyper-Threading Aware Process Scheduling Heuristics.” In: *USENIX Annual Technical Conference, General Track*. 2005.

[13] A. S. Chavan et al. “A comparison of page replacement algorithms.” In: *International Journal of Engineering and Technology* 3.2 (2011), p. 171.

[14] L. Clarke, I. Glendinning, and R. Hempel. “The MPI message passing interface standard.” In: *Programming environments for massively parallel distributed systems*. Springer, 1994, pp. 213–218.

[15] J. Corbalán and J. Labarta. “Improving processor allocation through run-time measured efficiency.” In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. IEEE. 2001, 6–pp.

[16] J. Corbalán, X. Martorell, and J. Labarta. “Performance-driven processor allocation.” In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. 2000.

[17] Intel Corporation. *8th Generation Intel Core Processor Families Datasheet, Volume 1 of 2*. `https://www.intel.la/content/www/xl/es/products/docs/processors/core/8th-gen-core-family-datasheet-vol-1.html`. Sept. 2018.

[18] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes:1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`. Oct. 2019.

[19] Intel Corporation. *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*. `https://software.intel.com/content/dam/develop/public/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf`. May 2019.

[20] Intel Corporation. *Intel Xeon Gold 6238R Processor Product Specification*. `https://ark.intel.com/content/www/us/en/ark/products/199345/intel-xeon-gold-6238r-processor-38-5m-cache-2-20-ghz.html`.

[21] Intel Corporation. *Intel Xeon Phi Processor 7200 Family Memory Management Optimizations*. Dec. 2016.

[22] Intel Corporation. *oneAPI Threading Building Blocks source code*. `https://github.com/oneapi-src/oneTBB/tree/v2021.2.0`. Version 2021.2.0.

[23] Intel Corporation. *Whitepaper: Energy-Efficient Platforms – Considerations for Application Software and Services*. Tech. rep. 2011.

[24] R. H. Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions.” In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[25] P. J. Denning. “The working set model for program behavior.” In: *Communications of the ACM* 11.5 (1968), pp. 323–333.

[26] P. J. Denning. “Virtual memory.” In: *ACM Computing Surveys (CSUR)* 2.3 (1970), pp. 153–189.

[27] A. Deshmeh, J. Machina, and A. Sodan. “Adept scalability predictor in support of adaptive resource allocation.” In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.

[28] E. W. Dijkstra. “Self-stabilization in spite of distributed control.” In: *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 41–46.

[29] X. Ding et al. “Bws: balanced work stealing for time-sharing multicores.” In: *Proceedings of the 7th ACM european conference on Computer Systems*. 2012, pp. 365–378.

[30] S. Dolev. *Self-stabilization*. MIT Press, 2000.

[31] A. B. Downey. *A model for speedup of parallel programs*. Tech. rep. University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, 1997.

[32] A. B. Downey. “Predicting queue times on space-sharing parallel computers.” In: *Proceedings 11th International Parallel Processing Symposium*. IEEE. 1997, pp. 209–218.

[33] H. Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling.” In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011, pp. 365–376.

[34] S. Eyerman, K. Du Bois, and L. Eeckhout. “Speedup Stacks: Identifying Scaling Bottlenecks in Multi-Threaded Applications.” In: *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS ’12. USA: IEEE Computer Society, 2012, pp. 145–155. ISBN: 9781467311434. DOI: 10.1109/ISPASS.2012.6189221. URL: https://doi.org/10.1109/ISPASS.2012.6189221.

[35] M. Z. Farooqui, M. Shoaib, and M. Z. Khan. “A comprehensive survey of page replacement algorithms.” In: *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume* 3 (2014).

[36] D. G. Feitelson and L. Rudolph. “Toward convergence in job schedulers for parallel supercomputers.” In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, pp. 1–26.

[37] LLVM Foundation. *LLVM OpenMP source code*. `https://github.com/llvm/llvm-project/tree/llvmorg-12.0.0/openmp`. Version 12.0.0.

[38] H. Franke, R. Russell, and M. Kirkwood. “Fuss, futexes and furwocks: Fast userlevel locking in linux.” In: *AUUG Conference Proceedings*. Vol. 85. AUUG, Inc. 2002, pp. 479–495.

[39] F. Freitag, J. Corbalan, and J. Labarta. “A dynamic periodicity detector: application to speedup computation.” In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. 2001. DOI: `10.1109/IPDPS.2001.924928`.

[40] D. Ghosal, G. Serazzi, and S. K. Tripathi. “The Processor Working Set and Its Use in Scheduling Multiprocessor Systems.” In: *IEEE Trans. Softw. Eng.* 17.5 (May 1991), pp. 443–453. ISSN: 0098-5589. DOI: `10.1109/32.90447`. URL: `https://doi.org/10.1109/32.90447`.

[41] O. Giersch. “Work stealing with dynamically sized thread pools.” MA thesis. BTU Cottbus–Senftenberg, 2019.

[42] C. Glawe. “Odtles: Turbulence modeling using a one-dimensional turbulence closed extended large eddy simulation approach.” PhD thesis. 2015.

[43] C. Glawe et al. “ODTLES simulations of turbulent flows through heated channels and ducts.” In: *Eighth International Symposium on Turbulence and Shear Flow Phenomena*. Begel House Inc. 2013.

[44] M. Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[45] A. Grama et al. *Introduction to parallel computing*. Pearson Education, 2003.

[46] J. L. Gustafson. “Reevaluating Amdahl’s law.” In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[47] T. Hagglund and K. J. Astrom. “PID controllers: theory, design, and tuning.” In: *ISA-The Instrumentation, Systems, and Automation Society* (1995).

[48] M. Hähnel et al. “Measuring Energy Consumption for Short Code Paths Using RAPL.” In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (Jan. 2012), pp. 13–17. ISSN: 0163-5999. DOI: `10.1145/2425248.2425252`. URL: `https://doi.org/10.1145/2425248.2425252`.

[49] H.-U. Heiss. “Dynamic partitioning of large multicomputer systems.” In: *Proceedings of the First International Conference on Massively Parallel Computing Systems (MPCS) The Challenges of General-Purpose and Special-Purpose Computing*. 1994, pp. 413–417. DOI: 10.1109/MPCS.1994.367051.

[50] M. D. Hill and M. R. Marty. “Amdahl’s law in the multicore era.” In: *Computer* 41.7 (2008), pp. 33–38.

[51] J. Huang, M. K. Qureshi, and K. Schwan. “An evolutionary study of Linux memory management for fun and profit.” In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 2016, pp. 465–478.

[52] T. Huang et al. “Extending Amdahl’s law and Gustafson’s law by evaluating interconnections on multi-core processors.” In: *The Journal of Supercomputing* 66.1 (2013), pp. 305–319.

[53] IEEE. *IEEE Std 1003.1-2017: POSIX.1a*. Web document. 2017. URL: https://pubs.opengroup.org/onlinepubs/9699919799/.

[54] IEEE. “Systems and Architectures.” In: *International Roadmap for Devices and Systems* (2020).

[55] T. Ilsche et al. “Powernightmares: The Challenge of Efficiently Using Sleep States on Multi-core Systems.” In: *Euro-Par 2017: Parallel Processing Workshops*. Ed. by D. B. Heras et al. Cham: Springer International Publishing, 2018, pp. 623–635. ISBN: 978-3-319-75178-8.

[56] Advanced Micro Devices Inc. *AMD EPYC™7003 Series CPUs Set New Standard as Highest Performance Server Processor*. https://ir.amd.com/news-events/press-releases/detail/993/amd-epyc-7003-series-cpus-set-new-standard-as-highest. Mar. 2021.

[57] Advanced Micro Devices Inc. *AMD64 Architecture Programmer’s Manual: Volumes 1-5*. https://www.amd.com/system/files/TechDocs/40332.pdf. Apr. 2020.

[58] Advanced Micro Devices Inc. *High Performance Computing (HPC) Tuning Guide for AMD EPYC™7003 Series Processors*. https://www.amd.com/system/files/documents/high-performance-computing-tuning-guide-amd-epyc7003-series-processors.pdf. Mar. 2021.

[59] Advanced Micro Devices Inc. *HPC Tuning Guide for AMD EPYC™Processors*. http://developer.amd.com/wp-content/resources/56420.pdf. Dec. 2018.

[60] UEFI Forum Inc. *Advanced Configuration and Power Interface (ACPI) Specification*. `https://uefi.org/sites/default/files/resources/ACPI_6_3_May16.pdf`. Jan. 2019.

[61] Texas Instruments. "Cmos power consumption and cpd calculation." In: *Proceeding: Design Considerations for Logic Products* (1997).

[62] I. A. Jabbie, G. Owen, and B. Whiteley. "Performance comparison of intel xeon phi knights landing." In: *SIAM Undergraduate Research Online (SIURO)* 10 (2017).

[63] Z. Jozefik et al. "One-dimensional turbulence modeling of a turbulent counterflow flame with comparison to DNS." In: *Combustion and Flame* 162.8 (2015), pp. 2999–3015.

[64] A. R. Kerstein. "One-dimensional turbulence: model formulation and application to homogeneous turbulence, shear flows, and buoyant stratified flows." In: *Journal of Fluid Mechanics* 392 (1999), pp. 277–334.

[65] K. Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements." In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3 (Jan. 2018). DOI: `10.1145/3177754`.

[66] G. Klein et al. "Comprehensive formal verification of an OS microkernel." In: *ACM Transactions on Computer Systems (TOCS)* 32.1 (2014), pp. 1–70.

[67] S. Kobbe. "Scalable and Distributed Resource Management for Many-Core Systems." PhD thesis. KIT, Karlsruhe, 2015. DOI: `10.5445/IR/1000048333`.

[68] S. Kobbe et al. "DistRM: distributed resource management for on-chip many-core systems." In: *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2011, pp. 119–128.

[69] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: a 32-way multithreaded Sparc processor." In: *IEEE Micro* 25.2 (2005), pp. 21–29. DOI: `10.1109/MM.2005.35`.

[70] R. Kuban. *PGAS for (In)coherent Manycore Systems*. Tech. rep. FG Verteilte Systeme und Betriebssysteme, 2021.

[71] R. Kuban and R. Rotta. *MyThOS Base Architecture ver. 2*. `https://github.com/ManyThreads/mythos/blob/master/doc/base-kernel/base-kernel.pdf`. Aug. 2016.

[72] C. Li, C. Ding, and K. Shen. "Quantifying the cost of context switch." In: *Proceedings of the 2007 workshop on Experimental computer science*. 2007, 2–es.

[73] Y. Li, Y. Matsubara, and H. Takada. "A Comparative Analysis of RTOS and Linux Scalability on an Embedded Many-core Processor." In: *Journal of Information Processing* 26 (2018), pp. 225–236. DOI: 10.2197/ipsjjip.26.225.

[74] C. Marcon et al. "Time and energy efficient mapping of embedded applications onto NoCs." In: *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. 2005, pp. 33–38.

[75] C. McCann, R. Vaswani, and J. Zahorjan. "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors." In: *ACM Transactions on Computer Systems (TOCS)* 11.2 (1993), pp. 146–178.

[76] F. T. Meiselbach. "Application of ODT to turbulent flow problems." PhD thesis. BTU Cottbus-Senftenberg, 2015.

[77] T. Mitra. "Heterogeneous Multi-core Architectures." In: *Information and Media Technologies* 10.3 (2015), pp. 383–394.

[78] I. Molnar. "Modular scheduler core and completely fair scheduler [CFS]." In: *Linux-Kernel mailing list* (2007). URL: https://lwn.net/Articles/230501/.

[79] D. Mulnix. *Intel Xeon Processor Scalable Family Technical Overview*. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html. July 2017.

[80] *musl libc*. https://www.musl-libc.org.

[81] *MyThOS source code*. https://github.com/ManyThreads/mythos.

[82] S. G. Narendra and A. P. Chandrakasan. *Leakage in nanometer CMOS technologies*. Springer Science & Business Media, 2006.

[83] T. D. Nguyen, R. Vaswani, and J. Zahorjan. "Parallel application characterization for multiprocessor scheduling policy design." In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, pp. 175–199.

[84] T. D. Nguyen, R. Vaswani, and J. Zahorjan. "Using runtime measured workload characteristics in parallel processor scheduling." In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, pp. 155–174.

[85] B. Oechslein. "Leichtgewichtige Betriebssystemdienste für ressourcengewahre Anwendungen gekachelter Vielkernrechner." doctoralthesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2018.

[86] B. Oechslein 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*. 2011, pp. 9–14.

[87] V. Pallipadi. “cpuidle - Do nothing, efficiently...” In: 2007. URL: `http://ols.108.redhat.com/2007/Reprints/pallipadi-Reprint.pdf`.

[88] V. Pallipadi and A. Starikovskiy. “The ondemand governor.” In: *Proceedings of the Linux Symposium*. Vol. 2. 00216. 2006, pp. 215–230.

[89] K. H. Park and L. W. Dowdy. “Dynamic partitioning of multiprocessor systems.” In: *International Journal of Parallel Programming* 18.2 (1989), pp. 91–120.

[90] P. J. Plauger. *The standard C library*. Prentice-Hall, Inc., 1992.

[91] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. " O’Reilly Media, Inc.", 2007.

[92] R. Rotta et al. “MyThOS — Scalable OS Design for Extremely Parallel Applications.” In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. 2016, pp. 1165–1172.

[93] Y. Ruan et al. *On the Effectiveness of Simultaneous Multithreading on Network Server Workloads*. Tech. rep. Department of Computer Science at Princeton University, July 2007.

[94] F. Schmaus et al. “System software for resource arbitration on future many-* architectures.” In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2020, pp. 967–975.

[95] C. Scordino, L. Abeni, and J. Lelli. “Energy-Aware Real-Time Scheduling in the Linux Kernel.” In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC ’18. Pau, France: Association for Computing Machinery, 2018, pp. 601–608. ISBN: 9781450351911. DOI: `10.1145/3167132.3167198`. URL: `https://doi.org/10.1145/3167132.3167198`.

[96] A. Shaik. “Shortest Time Quantum Scheduling Algorithm.” In: (2016).

[97] S. Siddha, V. Pallipadi, and A. Mallick. “Process Scheduling Challenges in the Era of Multi-Core Processors.” In: *Intel Technology Journal* 11.4 (2007).

[98] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts - International Student Version, 9th Edition*. Wiley, 2014. ISBN: 978-1-118-09375-7.

[99] K. Suo et al. “Quantifying context switch overhead of artificial intelligence workloads on the cloud and edges.” In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, pp. 1182–1189.

[100] M. B. Taylor. “Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse.” In: *DAC Design Automation Conference 2012*. 2012, pp. 1131–1136.

[101] M. B. Taylor et al. “Tiled Multicore Processors.” In: *Multicore Processors and Systems*. Boston, MA: Springer US, 2009, pp. 1–33. ISBN: 978-1-4419-0263-4. DOI: 10.1007/978-1-4419-0263-4_1. URL: https://doi.org/10.1007/978-1-4419-0263-4_1.

[102] Mellanox Technologies. *Product brief: TILE-Gx72 Processor*. Version rev 4.0. 2017.

[103] J. Teich et al. “Invasive Computing: An Overview.” In: *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Ed. by M. Hübner and J. Becker. New York, NY: Springer New York, 2011, pp. 241–268. ISBN: 978-1-4419-6460-1. DOI: 10.1007/978-1-4419-6460-1_11. URL: https://doi.org/10.1007/978-1-4419-6460-1_11.

[104] *The Linux Kernel Archives Website*. www.kernel.org. Version 5.4.

[105] *The OpenMP Website*. www.openmp.org.

[106] P. Thoman, P. Zangerl, and T. Fahringer. “Task-Parallel Runtime System Optimization Using Static Compiler Analysis.” In: *Proceedings of the Computing Frontiers Conference*. CF’17. Siena, Italy: Association for Computing Machinery, 2017, pp. 201–210. ISBN: 9781450344876. DOI: 10.1145/3075564.3075574. URL: https://doi.org/10.1145/3075564.3075574.

[107] *Top500 Supercomputer website*. https://www.top500.org/lists/top500/.

[108] A. Tucker and A. Gupta. “Process control and scheduling issues for multiprogrammed shared-memory multiprocessors.” In: *Proceedings of the twelfth ACM symposium on Operating systems principles*. 1989, pp. 159–166.

[109] J. Van Greunen and J. Rabaey. “Lightweight time synchronization for sensor networks.” In: *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. 2003, pp. 11–19.

[110] G. Varisteas and M. Brorsson. “Palirria: accurate on-line parallelism estimation for adaptive work-stealing.” In: *Proceedings of Programming Models and Applications on Multicores and Manycores*. 2014, pp. 120–131.

[111] S. Voinigescu. *High-Frequency Integrated Circuits*. Cambridge University Press, 2013.

[112] M. Voss, R. Asenjo, and J. Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. 2019.

[113] D. Wentzlaff and A. Agarwal. "Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores." In: *SIGOPS Oper. Syst. Rev.* 43.2 (Apr. 2009), pp. 76–85. ISSN: 0163-5980. DOI: 10.1145/1531793.1531805. URL: https://doi.org/10.1145/1531793.1531805.

[114] D. Wentzlaff et al. "On-chip interconnection architecture of the tile processor." In: *IEEE micro* 27.5 (2007), pp. 15–31.

[115] C. S. Wong et al. "Fairness and interactive performance of O (1) and CFS Linux kernel schedulers." In: *2008 International Symposium on Information Technology*. Vol. 4. IEEE. 2008, pp. 1–8.