Coscheduling in the Multicore Era

The Art of Doing Things Simultaneously

vorgelegt von
Dipl.-Inform.
Jan H. Schönherr

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Odej Kao
Gutachter:	Prof. Dr. Hans-Ulrich Heiß
Gutachter:	Prof. Dr.-Ing. Jan Richling
Gutachter:	Prof. Dr. Andreas Polze

Tag der wissenschaftlichen Aussprache: 19. Juni 2019

Berlin 2019

Kurzzusammenfassung

Der Begriff *Coscheduling* bezeichnet in seiner weitesten Definition die bewusst gleichzeitige Ausführung ausgewählter Tasks auf mehreren CPUs. Coscheduling und der verwandte Begriff *Gang Scheduling* wurden in den 80er bzw. 90er Jahren geprägt, als die ersten Mehrprozessorsysteme entwickelt wurden, bzw. sie weitere Verbreitung fanden.

Die garantierte gleichzeitige Ausführung von bestimmten Tasks erlaubt die effiziente Realisierung von feingranularer Synchronisation und Kommunikation, ohne dass es zu möglicherweise langen Wartezeiten kommt, weil auf einen gerade nicht laufenden Task gewartet werden muss. Im Vergleich zu beispielsweise Stapelverarbeitung oder der Partitionierung des Systems im Raum erlaubt es Coscheduling, die gerade laufende parallele Anwendung zu Gunsten einer anderen zu unterbrechen, wodurch Schedulingverfahren nun nicht mehr auf einzelnen sequentiellen Tasks operieren, sondern auf ganzen parallelen Anwendungen.

Seitdem hat sich die IT-Landschaft deutlich weiterentwickelt: aufgrund des Siegeszugs des (nicht parallelen) PCs ist die Thematik Parallelität zunächst in den Hintergrund getreten. Erst als weitere Taktsteigerungen nicht mehr möglich waren, wurde Parallelität wieder interessant. Im Vergleich zu früher gibt es allerdings zwei bedeutende Unterschiede:

1. Durch die vergleichsweise langsame Wiedereinführung von parallelen Systemen fand keine Neuentwicklung von Software statt, sondern eine Adaption. Damit werden aktuelle Mehrkernsysteme eher als verteiltes System denn als paralleles System behandelt: aktuelle Betriebssysteme kennen das Konzept einer parallelen Anwendung nicht; Coscheduling und andere Managementansätze sind ihnen fremd, worunter die Ausführung tatsächlich paralleler Anwendungen leidet. Zudem wird herangehenden Softwareentwicklern (außerhalb der HPC-Nische) die Entwicklung entsprechender Software selten nahe gebracht.

2. Aktuelle Mehrkernsysteme unterscheiden sich in ihrer Architektur stark von frühen parallelen Systemen und Clustern. Dies führt dazu, dass sich die Lösungen von damals nicht ohne weiteres auf heutige Systeme übertragen lassen. Inbesondere gibt es in heutigen Systemen diverse Ressourcen, die sich mehrere CPUs teilen müssen, wie z. B. Speicherbandbreite oder Caches. Dies führt zu Ressourcenengpässen, die zu mehr oder weniger starken Leistungseinbußen individueller Tasks führen können. Die Forschung

schlägt hier zumeist ein geschicktes Gruppieren von Tasks vor, so dass Ressourcenengpässe nach Möglichkeit minimiert werden – im Prinzip eine andere Form von Coscheduling. Während jeder Vorschlag für sich seine Daseinsberechtigung hat, ist eine Integration in bestehende Systeme meist unpraktikabel, da der Fokus oft eingeschränkt ist und die verschiedenen Ideen in Konflikt miteinander stehen.

In dieser Dissertation wird das Konzept *Coscheduling* auf heutige Mehrkernsysteme übertragen und adaptiert. Die früher gültigen Vor- und Nachteile werden neu bewertet und alte wie neue Anwendungsszenarien betrachtet. Aus einem Anforderungskatalog heraus wird ein Coschedulingverfahren konzipiert, das den heutigen Erwartungen von Anwendungs- und Betriebssystementwicklern gerecht wird. Es wird eine Methode dargelegt, wie besagtes Verfahren in bestehende Betriebssysteme integriert werden kann, ohne dass es zu einer wesentlichen Verhaltensänderung des ursprünglichen Schedulers kommt. Es wird gezeigt, welche neuen Managementmöglichkeiten sich dadurch für das Betriebssystem ergeben und welchen Einfluss dies in Zukunft auf die Entwicklung paralleler Anwendungen und Betriebssysteme haben wird.

Abstract

In its most general definition, *coscheduling* refers to a deliberate simultaneous execution of certain tasks on multiple CPUs. The concepts of coscheduling and gang scheduling have been introduced in the early eighties and nineties, respectively. At that time, the first massively parallel systems were developed and became more widely used.

The guarantee of simultaneous execution of certain tasks allows to make use of fine-grained synchronization efficiently: it is impossible for a currently executing task to wait for another task that does not make progress. Compared to batch processing and partitioning in space, coscheduling allows to preempt a running parallel application in favor of another more important application. This context switch at application level offers more flexibility for schedulers.

Since then, there have been thorough changes in the computing landscape: (non-parallel) personal computers have penetrated the market, and with it, parallelism had become a second-class citizen. Parallelism has only recently become important again, after single-thread performance could not be increased by the usual margins anymore. However, parallelism today differs from parallelism back then, mostly because of the following two reasons:

1. The reintroduction of parallel systems happened not overnight but as a gradual process. Due to this, existing software was adapted to run on parallel systems instead of being rewritten. This adapted software handles contemporary multicore systems as if they were distributed systems instead of the parallel systems they are: current operating systems have no concept of a parallel application; they do not know about coscheduling or other management approaches, hampering real parallel applications. To make matter worse, budding software developers (outside of the HPC niche) are seldom taught the subtleties of parallel software development.

2. The properties of today's multicore architectures differ substantially from early parallel systems and clusters. For this reason it is not always possible to reuse once valid solutions. In particular, CPUs in today's system have to share a multitude of resource, such as memory bandwidth of caches. This results in resource contention with a more or less noticeable impact on performance of individual tasks. In most cases research suggests to avoid or reduce resource contentions by grouping tasks skillfully – which is basically a form of coscheduling. However, an integration of these research

ideas into existing systems is impractical more often than not, because individual ideas – while solving their specific use case – are difficult to combine with their rather narrow focus.

This thesis is ports the concept *coscheduling* to contemporary multicore architectures. Advantages and disadvantages known from other parallel architectures are reevaluated, and a catalog of use cases – old and new – is compiled. The combined requirements of these use cases form the basis for a versatile coscheduling model. A flexible coscheduling approach is devised, which measures up to today's expectations of application and operating system developers alike. In particular, a method is included that allows to integrate coscheduling functionality into existing general purpose schedulers without changing their characteristic traits substantially. Newly enabled management opportunities within the operating system are discussed and their influence on the design of upcoming parallel applications and operating systems are explored.

To Cindy.

Without her I would not have embarked on this journey.

Contents

Chapter 1

Introduction

The scheduler of an operating system is responsible for managing the CPUs of a computer system, assigning them to the different tasks present in the system. This generation of a schedule within a dynamic system is usually nontrivial as different goals, such as performance or energy efficiency, collide. Even when a goal is finally set, the problem remains NP-complete and might require knowledge that is unavailable in general purpose computer systems, such as future behavior of tasks. Hence, scheduling usually relies on heuristics to operate somewhere near the intended goal.

In multicore systems the resource CPU is not unique anymore, complicating the management of this resource. Today, CPUs are handled as anonymous resources, allocated and deallocated to tasks by the operating system scheduler on every context switch. In general, there is neither a direct control over this allocation process available for applications, nor some kind of notification mechanism when the operating system modifies the resource allocation. Applications can request CPUs only indirectly by creating one or more tasks. (Though, most operating systems allow applications to bind tasks to specific CPUs.) This lack of control within applications allows the operating system a great deal of flexibility to reach its scheduling goal, as there are (almost) no external constraints on task placement. For a wide range of applications, this lack of control and guarantees does not pose a problem. One notable exception are real-time tasks, where too little CPU time leads to missed deadlines, which can have disastrous results in the physical world. Another exception is the class of parallel applications, which can suffer extreme performance losses due to non-parallel execution.

To efficiently support all application classes, the control of the resource CPU must be made more explicit. However, more sophisticated approaches to manage CPUs are rarely found. And some more wide spread management concepts, such as explicit control over CPU affinity, are next to useless unless they are coordinated system wide. In a parallel application, for example, using explicit CPU affinities avoids running more than one task of that application on the same CPU; however, it does not provide any guarantees regarding simultaneous execution of tasks or exclusive access to CPUs. A solution for the problematic

classes would be to explicitly allocate and free CPUs. But this poses too severe restrictions on possible schedules for general purpose computing systems with multiple applications. Therefore, a more operating system friendly solution lets applications request certain scheduling guarantees, which are then honored by the operating system scheduler. This helps applications, which can now rely on a certain behavior, and it helps the operating system, which has more degrees of freedom with that. Such guarantees could be, for instance, a task getting a certain amount of CPU time within a certain period (for real-time tasks), or a set of tasks always being executed simultaneously (for parallel applications). The latter guarantee is also known as coscheduling or gang scheduling, which is the topic of the remaining thesis.

1.1 The Significance of Coscheduling

Coscheduling – roughly speaking the simultaneous execution of selected sets of tasks – was introduced on early parallel systems [1]. This "context switch at system level" basically allows multiprogramming of parallel machines, while each application still sees the whole machine as its own. Among other things, this allows applications to use fine-grained communication efficiently. Because communicating tasks are executed simultaneously, their processing is not delayed by unrelated tasks and timely answers are guaranteed. Especially, it is possible to use busy waiting to synchronize tasks, without burning CPU cycles waiting for tasks that are not even running. But coscheduling is not only useful at application level. In current research, it evolves into a tool, that is exploited by the operating system itself to optimize the usage of shared resources, by considering which tasks should execute simultaneously.

Unfortunately, coscheduling still carries the stigma of being not scalable, costly to realize, and generally not worth the effort, because of fragmentation problems and not enough advantages to be gained from common applications. And indeed, none of the current general purpose operating systems has implemented something even remotely similar to coscheduling.

However, the hardware and software landscape has changed substantially in the last decade and is still changing. And with this, the relevance of coscheduling changes. Today, parallel applications are seldom created from scratch. Instead, they are usually developed on top of some parallel substrates, such as *OpenMP* [2] or the *Intel Threading Building Blocks* [3], which encapsulate a lot of the necessary parallel programming expertise. This causes more and more applications to be parallel, as these environments can also be used by developers, who are less experienced in parallel programming, enabling them to produce sensible results. A consequence of this is, that the question whether an application would benefit from coscheduling has not to be answered for each individual applications anymore, but instead just for a few parallel programming substrates. These few substrates could be changed without too much time and effort to exploit the benefits of being coscheduled.

In every new processor generation, each CPU shares more and more resources with other CPUs. For a processor this means a better utilization of its transistors; for a single task it means potentially less performance due to other tasks "stealing" resources. While every new hardware generation still delivers more performance than the one before it, the performance gap between a non-optimized schedule and an optimized schedule also increases. While most operating system schedulers have learned to place tasks somewhat intelligently within mostly idle systems to reduce the resource contention caused by non-optimal schedules, they do not go further than this. By not only controlling *where* a task is executed, but also *when* it is executed in relation to other tasks within the system, it is possible to close this gap again. Having to deal with an increasing amount of parallel applications adds another layer of complexity.

1.2 Problem Statement

With respect to parallel applications, development of operating system schedulers has stagnated. Schedulers consider tasks only individually; they have no notion of parallel applications. Bad scheduling behavior on new architectures (e. g., when simultaneous multithreading (SMT) became mainstream) is just flimsily fixed and not addressed thoroughly. Though, this is not due to a lack of ideas. Rather, individual implementations of new ideas to solve specific issues are often not fit for the requirements of general purpose operating systems, or they obliterate any chance of peaceful coexistence with other scheduler improvements, reducing possible benefits and applicability.

Coscheduling has the potential to further development on all those fronts: parallel applications can profit from simultaneous execution, system design can be improved or simplified by utilizing coscheduling, and resource management can be optimized by coordinating scheduling across multiple cores. However, coscheduling is also known as a "scalability nightmare waiting to happen" as it was described by Peter Zijlstra, one of the Linux scheduler maintainers, on the Linux Kernel Mailing List (LKML) in April 2010. A sentiment that is shared by others.

This thesis reevaluates the concept of coscheduling on contemporary multicore systems and addresses the main drawbacks associated with coscheduling. It answers the following questions:

1. What are the functional requirements that are placed on a coscheduler for general purpose multicore systems?

2. How can such a coscheduler be realized without falling victim to non-functional shortcomings?

3. Which hitherto unknown applications of coscheduling are possible on multicore systems?

1.3 Contribution

By answering the aforementioned questions, this thesis makes the following primary contributions to the state of the art:

1. A model for coscheduling is developed that is sufficient for all currently known use cases.
2. A coscheduler design is derived from functional requirements, that exceeds existing designs in terms of flexibility and versatility.
3. A method is provided that integrates said design into existing general purpose schedulers without changing their characteristic traits.
4. The foundation of a management concept is presented that unifies previously independent domains of coscheduling research.
5. Several new use cases for coscheduling are identified.

Together, these contributions should remove any inhibitions of scheduler developers to integrate coscheduling into their product. This in turn paves the way for many use cases to make the transition from research (or niches) to real life.

Furthermore, the following secondary contributions are made:

1. The first cross-domain survey of coscheduling use cases on multicore architectures is made.
2. The generally accepted definition of coscheduling is slightly generalized.

1.4 Structure of this Thesis

This thesis is split into three parts, moving from foundations to design to the application of coscheduling.

The first part covers the foundations of coscheduling. First, Chapter 2 gives a thorough survey of coscheduling use cases, covering all use cases discovered to date as well as new ones discovered by the author while preparing this thesis. These use cases are then dissected in Chapter 3 to develop a model for coscheduling and to introduce a common terminology. Chapter 4 finishes the foundation by taking a look at existing coscheduling approaches and presenting them in light of the newly gained insights.

The second part is dedicated to the creation of a coscheduler design suitable for contemporary multicore systems allowing to realize the various use cases. Design rationales driving the creation are motivated in Chapter 5, before the resulting design itself is described in Chapter 6. Its distinctive feature compared to other coscheduling solutions is its topology-awareness, which can refer to both,

the organization of hardware components and the logical structure of software. Hence, the design is also referred to as Topology-Aware Coscheduling (TACO). After that, Chapter 7 focuses on ways to realize TACO in already existing schedulers without breaking their distinctive features. It gives two examples: Linux and FreeBSD; with a specific focus on the first one. The part closes with Chapter 8, where basic aspects of TACO are evaluated to demonstrate the fulfilment of the design rationales.

The third part of the thesis addresses complex scenarios, where coscheduling is applied. Chapter 9 compares different solutions for managing the simultaneous execution of many parallel programs. In this scenario, that will likely be commonplace in a few years, a solution based on TACO is able to outperform established approaches in most situations with the added benefit of being backwards compatible to applications that are not specifically tailored to a managed scenario. Chapter 10 demonstrates how coscheduling can be applied in recent processors to make the automatic distribution of the energy budget (a. k. a. turbo boost) more efficient: in scenarios with jobs of varying importance, performance is improved consistently, while the energy consumption even gets reduced in certain cases. The last chapter in this part, Chapter 11, details new design ideas and design alternatives for some traditional operating system functionalities based on coscheduling. Specifically, a concept based on TACO is presented that allows to reuse advanced scheduling optimization techniques – originally developed for sequential applications – with parallel applications without having to redesign said techniques. Additionally, the concepts of mutexes, affinities, and concurrency control are revisited and advantages and disadvantages of their realization with coscheduling are discussed.

The thesis closes with Chapter 12, where the prospects of coscheduling are assessed, taking into account the results of this work as well as current trends in hardware design and software development.

1.5 Chapter Notes

In addition, each chapter ends with chapter notes similar to this one. In them, you will find some of my personal thoughts pertaining to each chapter. These can range from additional facts or anecdotes to a more subjective view on the chapter's contents. They provide a chance to reflect on the presented contents, before the thesis moves on to another topic in the next chapter.

Part I

Coscheduling Foundation

where use cases are presented, a model for co-scheduling is developed, and existing approaches are surveyed

Table of Contents

Chapter 2

Why Coscheduling?

This chapter gives a thorough survey of use cases in which coscheduling has been found useful on multicore architectures. This chapter focuses on *why* coscheduling is beneficial in certain scenarios. Different approaches *how* coscheduling can actually be achieved are discussed later in Chapter 4.

Coscheduling can be applied with different goals in mind (see Fig. 2.1). Usually these are optimizations of some kind: optimization of performance, energy consumption, or energy efficiency. But there is also the goal to improve the design of a piece of software, e. g., make it simpler, less error-prone, or more secure. Table 2.1 gives an overview of all use cases presented in this chapter.

This chapter organizes the different coscheduling use cases by their scope at which these use cases are usually applied. Section 2.1 presents use cases focusing on individual (parallel) applications. Section 2.2 contains uses cases which optimize the usage of resources of typically independent tasks. As such, these use cases are usually applied at the operating system level, though parallel applications may also benefit. Finally, Section 2.3 surveys use cases which target specifically the operating system and its interfaces.

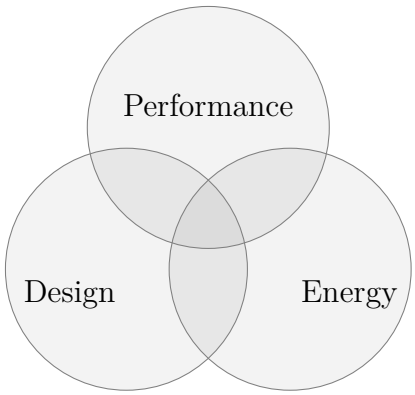


Figure 2.1: Areas, in which coscheduling can induce benefits.

Table 2.1: Summary of Coscheduling Use Cases.

Use case	Improves	For
Application Design		
Fine-grained synchronization	Design	Application
Scalability of algorithms	Design	Application
Response time	Performance	Application
Lock holder preemption	Performance	Application
Static load balancing	Design	Application
Auxiliary tasks	Design	App./System
Execution unit optimizations	Performance	Application
Cache optimizations	Performance	Application
Resource Management		
Memory pressure	Performance	Application
Cache pressure	Performance	Application
Execution unit contention	Performance	Application
Cache/memory bandwidth contention	Performance	Application
Temperature balancing	Energy/Perf.	System/App.
Power capping	Energy/Perf.	System/App.
Energy contention	Energy/Perf.	System/App.
System Design		
Parallel application management	Design	System
Concurrency management	Design/Perf.	System/App.
Affinity management	Design	System
Side-channel elimination	Design	System
Exclusive device access	Design/Perf.	System/App.
Expose system functionality	Design/Energy	System
Special architectures	Design	Hardware

(a) Active waiting.

(b) Passive waiting.

Figure 2.2: Two tasks repeatedly entering critical sections in an otherwise idle system. The active waiting variant has a performance advantage over time due to the extra time needed to wake up sleeping tasks.

2.1 Application Design

This section contains use cases for coscheduling, which have a noticeable effect on the design of applications – usually simplifying the design of an application in some way. None of the presented use cases have a negative impact on application performance. When applicable, alternative design options to reach the same goal are discussed.

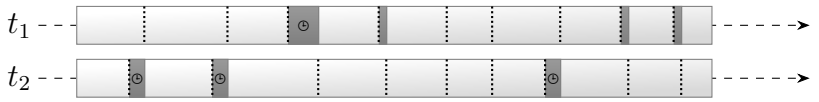
2.1.1 Active Waiting

Waiting for something is a building block found often in software engineering: waiting for a lock to become free, waiting for a message to arrive, waiting for a result to become available, etc. Waiting comes in two flavors: active waiting (a. k. a. busy waiting or spinning), where the CPU actively and repeatedly checks for the occurrence of an event, and passive waiting (a. k. a. blocking), where the current task is suspended until the event in question is signalled by another component (e. g., another task or a hardware interrupt). Active waiting is usually considered as something that should be avoided, because it prevents the CPU from doing useful work. However, there are two situations where the use of active waiting over passive waiting is indicated:

1. The expected waiting time is less than the expected overhead of passive waiting, which not only includes context switches but also cache misses due to the aforementioned context switches and tasks that were executed in the meantime.

2. There is nothing else to do, and the delay of entering and leaving a power save mode is not acceptable. (Or there is no mechanism in place to realize passive waiting.)

In an otherwise idle system, active waiting allows to avoid unnecessary user/kernel-space transitions and prevents delays from entering and exiting a



(a) Active waiting.

(b) Passive waiting.

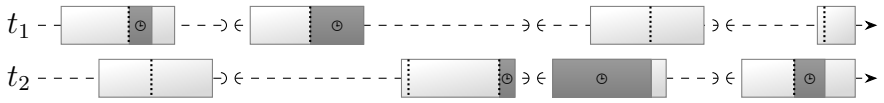
Figure 2.3: Two tasks repeatedly doing barrier synchronizations in an otherwise idle system. The active waiting variant has less overhead.

deeper sleep state of the CPU. While the saved time is rather small for a single occurrence, it can accumulate to measurable differences. This is illustrated in Fig. 2.2, where two tasks repeatedly enter very short critical sections, e. g., to update a small lock protected data structure.

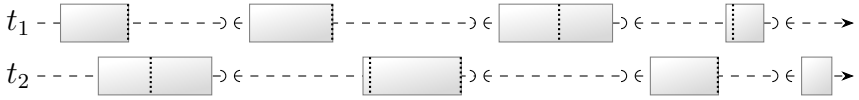
The figure shows the timeline of two tasks. As in other figures, time moves from left to right and each horizontal line usually shows a separate task and its behavior while it is running. (Later figures may show individual CPUs and which tasks they are currently executing.) Here, a lock symbol indicates that a task is within a critical region; a small clock denotes a task that is actively waiting – in this case to enter the critical region. In the case of passive waiting, the overhead of putting the CPU of a task to sleep and waking it up again is hinted at by dark grey bars.

If wait times are rather small, entering a deeper sleep state is also counterproductive from an energy perspective, as CPUs might more often be required to wake up while still transitioning into a sleep state. An example for this is given in Fig. 2.3, with two tasks repeatedly doing barrier synchronizations. The more balanced the work is, the more overhead is induced by passive waiting.

Today, most environments are multiprogrammed, and applications usually have no control over when and where they are executed. Specifically, a task can be preempted at any time. This makes active waiting at application level problematic, because whatever event a task is waiting for, the generating task can be delayed arbitrarily. That is, it is impossible to predict the expected waiting time adequately. In the worst case, the actively waiting task is scheduled on the very same CPU as the event generating task – essentially blocking its own progress. Such unexpected delays with active waiting are illustrated in Figure 2.4, which shows two tasks that repeatedly perform barrier synchronizations, e. g., to signal their readiness to begin the next round of a parallel algorithm. In addition to these tasks, there is also other load in the system, so that the displayed tasks cannot run all the time. The short breaks in the timeline indicate possible skips in time where solely other load is executed.



(a) Active waiting.



(b) Passive waiting.

Figure 2.4: Two tasks repeatedly doing barrier synchronizations in a system that also schedules other tasks. The active waiting variant causes considerable overhead, which gets worse with less time-slice overlap and shorter intervals between barriers.

Figure 2.5: Same situation as in Fig. 2.4, but this time with active waiting and coscheduling. The overhead of active waiting is contained and there are less context switches than in the passive waiting case.

Fine-grained Synchronization

Coscheduling makes active waiting in multiprogrammed scenarios efficient again. In its simplest form, coscheduling gives an application the guarantee of synchronous execution of its tasks. Thus, when a task is actively waiting for an event generated by another task, it is guaranteed, that this other task is currently being executed. Furthermore, should the other task get preempted for some reason, the actively waiting task will also be preempted, so that no CPU cycles are unnecessarily burnt. Figure 2.5 shows the previous scenario, but this time the communicating tasks are coscheduled. This is now almost as if the communicating tasks are executed in isolation. The one difference (which is not properly reflected in the diagrams) is that every time a task is scheduled, the cache will have to be refilled, which causes additional slow downs compared to an isolated execution. Note, however, that the coscheduled case results in fewer context switches than in the non-coscheduled passive waiting case.

In the end, it depends on how fine-grained the synchronization between tasks actually is, whether uncoordinated passive waiting or coscheduled active waiting is the more efficient solution. The more fine-grained it is, the smaller is the average waiting time and the more context switches are saved by coscheduled active waiting. The cross-over point was analyzed in more detail by Feitelson and Rudolph in [4]. Though, they did not consider the effect of caches: today's context switch costs are dominated by indirect costs due to cache misses as

shown by Liu and Solihin in [5] for sequential applications. This line of thinking is picked up later in Chapter 8 for parallel applications, where an upper bound for the overhead of coscheduling compared to isolated execution is derived.

For parallel applications or algorithms that already make use of fine-grained synchronization, for instance, because they were thought to be executed in isolation, coscheduling allows to execute them nearly without performance degradation in multiprogrammed environments. This mostly applies to two scenarios: software originally developed for high performance computing (HPC) environments and then reused on multiprogrammed systems without adjusting algorithms properly, and software originally executed on dedicated machines and then migrated to virtualized environments, where – while individual VMs may still be dedicated – multiple VMs are executed on the same system. A third scenario is the execution of software in environments that are not as isolated as they ought to be. This is seen for instance in HPC environments, where activities of the operating system (OS) – so called OS jitter – interrupt and delay the execution of the currently processed parallel application. While each individual interruption is rather short, especially large scale collective communications suffer from this. Coscheduling can be used here to achieve the desired performance as demonstrated by Petrini et al. in [6].

Scalability of Algorithms

For application developers, the prospect of efficient fine-grained synchronization opens the possibility to do a finer problem decomposition than they would have done otherwise. Thus, smaller problems become parallelizable and decompositions may contain more dependencies than usual. Algorithms can be scaled to larger machines, because subtasks may be smaller.

Response Time

A different issue is the response time of parallel applications in multiprogrammed scenarios, which might suffer with passive waiting, because of additional delays between waking up a task (i. e., inserting it back into the runqueue) and the actual execution of that task. How long that delay is, strongly depends on the load of the system and the employed scheduling strategy. Usually, schedulers give interactive tasks some kind of preferential treatment, which – at least theoretically – would keep response times low. However, simple classification strategies might mistake non-interactive parallel applications with fine-grained synchronization as interactive, when they use passive waiting, due to their short processing phases. This would nullify any improvement of response times of truly interactive applications. (This is demonstrated in a slightly different context by Xu et al. in [7].) In such a situation, even applications with passive waiting may profit from being coscheduled.

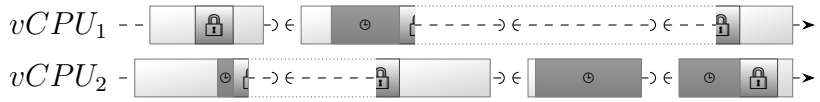


Figure 2.6: Lock holder preemption. Although every lock is only held for a short period in terms of task runtime, a preemption can prolong the actual hold time arbitrarily.

Lock Holder Preemption

The currently most prevailing problem, however, are virtual machines with multiple (virtual) CPUs, i. e., SMP guests executing on SMP hosts. Fully virtualized SMP guests are a prime example for a piece of parallel software that can not be redesigned so that is does not make use of active waiting: practically every OS for multicore systems makes use of spinlocks for low-level synchronization. On real hardware, this is not a problem as a lock is only held for a very short amount of time. In a virtualized environment, however, a CPU of a guest – a so-called *virtual CPU (vCPU)* – is represented as a thread in the host system, which can be preempted at any time. The effect has been aptly named *lock holder preemption (LHP)*, which is illustrated in Fig. 2.6. (Strictly speaking, the term LHP could also be used for most of the previously given examples when using the more general definition of *lock* as known from lock-free algorithms [8].) With the current usage of virtualization in cloud computing environments, LHP has become a real problem, with lots of people trying to alleviate it in one way or the other.

Practically used solutions to defuse LHP in virtualization settings include coscheduling in VMware vSphere [9], paravirtualized spinlocks in Linux [10], and pause loop exiting (PLE) in Intel processors [11] or pause filtering (PF) in AMD processors [12]. Other techniques include delayed preemption [13], preemptable ticket locks [14], and lock-free operating systems [15]. Table 2.2 gives a short overview over these approaches. Without going into detail, each approach has its drawbacks: All except coscheduling, PLE/PF, and delayed preemption via observation require modifications of the guest OS, which are not always possible. Preemptable ticket locks, while being a guest-only solution, do not fully solve LHP (ticket locks are merely degraded to regular spinlocks, when the next task in the queue is not running). Delayed preemption must also be protected against abuse by a guest; the variant with observation is also rather coarse-grained as it makes decisions on whether a vCPU is in kernel mode or user mode. Paravirtualization (PV) of spinlocks as done in Linux results in passive waiting semantics – which includes the response time problem mentioned above. The hardware assisted solutions PLE/PF are semantically equivalent to spin-blocking (waiting actively for a moment before blocking), but contrary to

Table 2.2: Approaches to handle LHP. Many of these can also be applied outside of virtualization settings – then host and guest have to be substituted by OS and application, respectively.

Approach	Modifications	Result
Coscheduling	host only	useless waiting avoided
Paravirtualized spinlocks	host, guest	waiting avoided
Pause loop exiting/ pause filtering	host, hardware	waiting truncated, requires virtualization
Delayed preemption (via PV)	host, guest	LHP largely avoided
Delayed preemption (via observation)	host only	LHP largely avoided, requires virtualization
Preemptable ticket spinlocks	guest only	waiting partly avoided
Lock-free algorithms	guest only	LHP non-existent

spin-blocking they do not convey information about unblocked vCPUs, which requires guesswork in the host to decide which vCPU should be executed next. Coscheduling and PLE/PF are the only solutions that also work, when the actual lock holder is not the guest OS itself but a task of a parallel application within the guest (assuming that LHP within the guest itself is not a problem). This makes coscheduling the only generally applicable solution, when not employing lock-free algorithms – which are still elusive for most developers – in all guests.

While the guarantees provided by coscheduling might be too strong for some applications, it should also be noted that this form of LHP handling requires almost no information about the executed applications at system level. It is sufficient to specify groups of tasks which might wait for each other. However, with additional information coscheduling can be applied more fine-granular, which makes this solution also suitable for applications which do not profit from being coscheduled otherwise. This is explored later in Chapter 11.

2.1.2 Synchronous Execution

The guarantee of synchronous execution that is provided by coscheduling is quite strong. While it enables efficient fine-grained synchronization, it also provides benefits in other scenarios. Particularly, synchronous execution of tasks implies that these make progress equally fast. This lesser guarantee is useful in the area of load balancing. The full guarantee is needed to realize certain classes of auxiliary tasks.

Static Load Balancing

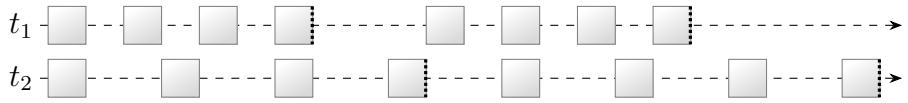
In order to make full use of a dedicated parallel machine, it is good practice to balance the computational load evenly across all available CPUs. This can be either done dynamically at runtime or statically already at design time [16]. Which one should be preferred, depends on several different factors. Dynamic load balancing has the advantage of flexibility. It can handle almost any situation, especially situations with varying characteristics, such as subtasks of varying complexity. However, it comes with overhead at runtime: calculation of load, handling of work queues, and possibly migration of subtasks. Static load balancing is less flexible, but has no overhead at runtime. In the best case, there is no need for communication between CPUs: each CPU just processes its part of the problem. Static load balancing is for instance practiced in the HPC area with MPI [17]. Another example is OpenMP [2], where iterations of a parallel loop are also distributed statically between tasks by default. While not really static, parallel virtual machines (VMs) can be seen as another example: the scheduler of the guest OS usually assumes that each of its vCPUs delivers constant computational power, and balances load accordingly.

If the parallel system is also used for unrelated computations, static load balancing becomes a burden: every unrelated computation on a CPU causes a delay. At the end of the parallel job, most tasks will have finished, except for those that have been delayed. This happens, because OS schedulers usually only provide a guarantee of fairness and not a guarantee of equal progress. Hence, even small, transient disparities in fairness may lead to a large difference in received CPU time, eventually. And scheduling algorithms normally do not give tasks a chance to catch up CPU time. With coscheduling, equal progress is enforced and static load balancing can be used within parallel applications for multiprogrammed environments as shown in Fig. 2.7.

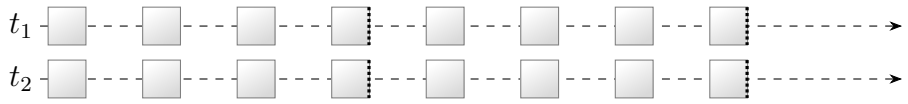
Auxiliary Tasks

The guarantee of synchronous execution also opens a new design option in multiprogrammed scenarios: auxiliary tasks that augment the functionality of the main task in some way. Contrary to parallel algorithms, where multiple tasks work cooperatively on a problem, auxiliary tasks are more or less optional. Potential techniques for auxiliary tasks include prefetching [18, 19], speculative execution [20], and analysis [21, 22].

Prefetching auxiliary tasks are tasks that prefetch memory for the main task into a shared cache, so that the main task experiences as few stalls as possible. This is beneficial, when the data access pattern is irregular and not detected by hardware prefetchers (e. g., pointer chasing). With task-level speculation auxiliary tasks start with some calculations early, without being sure that their input values are the correct ones. Ideally, it turns out that the input values have not changed, when the main task finally requires the results of the speculative execution. If a misspeculation is detected, the results of an auxiliary task are simply



(a) Without coscheduling.



(b) With coscheduling.

Figure 2.7: An application with two tasks and static load balancing in a system with an odd number of tasks. Without coscheduling, the imbalance will cause one task to make less progress than the other; and it strongly depends on the synchronization frequency within the application and a potential rebalancing interval, whether the uneven progress has a chance to even itself out or not. With coscheduling, the imbalance is still there but does not affect the application anymore.

discarded and the calculation is redone. Finally, auxiliary tasks can be used for analysis instead of performance improvements. The advantage of realizing the analysis of a main task with separate tasks is that the main task does not need to have costly instrumentation, i. e., it executes with its original performance despite being analysed. The analysing can be direct, such as monitoring modifications of data structures and stack, or indirect, such as looking for behavioral changes when shared resources are more or less stressed by one or more auxiliary tasks.

The synchronous execution provided by coscheduling allows to use these techniques also in multiprogrammed scenarios, where tasks compete for computational resources. Most of these techniques also depend on properties of the underlying system, and thus require a certain placement relative to the main task within the parallel system. Prefetching requires a shared cache, speculation is likely to profit from a shared cache, and analysis requires sharing of resources depending of what actually is analysed.

Specifically for the purpose of analysis, coscheduling also provides the guarantee, that other tasks will not be able to influence the analysed task. That is, coscheduling provides a controlled environment in a dynamically used system. Prefetching and speculation trade computational resources for a bit more performance. Depending on the effectiveness, it might make sense from a system perspective not to employ these techniques, when enough load is available within the system. These trade-offs are discussed in Chapter 9. A mechanism to enable the operating system to make such decisions, which can also be applied to task pools in general, is explored later in Chapter 11.

2.1.3 Architecture-specific Optimizations

One facet of software optimization is to tune software towards the system or systems where it is going to be deployed, so that the software is able to utilize the target system to its fullest potential. Today, this is particularly done in the HPC area and in multimedia applications. (In some sense, VMs are also highly optimized. The difference is, that they do all their optimization towards the target system at runtime.)

While a tremendous amount of target specific optimization is done by the compiler itself, the compiler cannot do everything. Especially, automatic parallelization is – except for special cases – an as of yet mostly unsolved problem. Usually the developer is responsible for high-level optimization and parallelization, while the compiler covers low-level aspects – though the boundaries are blurred. In the end, it is a symbiotic process with the developer providing suitable input, which is then turned into something that will execute efficiently on the target system.

With respect to multicore architectures and coscheduling, two optimization techniques are of particular interest: optimization for execution unit occupancy and optimization of cache utilization. Both target resources, which are potentially shared between CPUs. For these and also other shared resources, optimization differs depending on whether you do or do not know, what other CPUs are executing. Coscheduling gives parallel applications this knowledge, no matter what else is executed in the system. Thus, coscheduling makes it possible to use established methods of parallel application optimization without risking that these optimizations might work against them as it would be possible without coscheduling.

On the other hand, if it turns out that tasks of a parallel application are simply a bad match for a certain shared resource, a coscheduling capable operating system will also be able to avoid coscheduling of these tasks to some extend and find better tasks to execute them with (see also Section 2.2).

Execution Unit Optimizations

If CPUs share some of their execution units with other CPUs, i. e., the system has support for simultaneous multithreading (SMT), this should be considered by optimized parallel applications. Consider, for example, a system with processors similar to AMD's Bulldozer microarchitecture [23], where each CPU has its own integer execution unit but has to share a floating point execution unit with a second CPU. Assume further, that in this system each execution unit is able to finish one instruction per clock cycle. An optimized single-threaded application will issue one integer and one floating point operation per cycle to fully utilize the execution units of its CPU. This is ideal, when the shared floating point unit is not used by the other CPU. However, imagine a task optimized in this way within a parallel application: two tasks would now compete for the floating point unit, making it the bottleneck. This in turn causes a reduced utilization of the

integer units. Within a parallel application on this particular system, it would be ideal for every task to issue one integer operation every cycle and one floating point operation every second cycle. This would result in a fully utilized system with no particular bottleneck. (This assumes that tasks do similar things. There is also the option to make use of execution units asymmetrically. For the CPU described above, this would be one task with only integer operations for every task optimized as in the sequential case.)

Cache Optimizations

Cache optimizations are well known from sequential applications. They are important because of the large performance difference of cache and main memory and the potential improvements that can be achieved. Most techniques target the data processing loops of applications, modifying data traversal and access patterns to be more suitable to the target architecture. With the polyhedral model exists an approach to solve certain cases programmatically (e. g., [24]). Another approach are auto-tuning mechanisms, where several variations of an algorithm are executed to determine the best variant experimentally (e. g., [25]).

Multicore architectures, where caches are shared between CPUs, add another level of complexity to cache optimizations: the performance of a task no longer depends just on its own handling of the cache; what tasks on other CPUs are doing in the shared cache will also affect the task's performance. Thus, parallel applications must take this into account and distribute their work, so that tasks running on CPUs with shared caches always work on similar parts of the problem (e. g., [26]). Once the algorithm for that is in place, the finer details can be again determined by auto-tuning. The involved processes are demonstrated using the example of parallel stencil computations in [27].

A different approach to handle cache optimizations are cache oblivious algorithms [28], which are designed to work quite well with whatever amount of cache is available without the need to tune any parameters: they usually work on progressively smaller amounts of data, which will fit into the cache at some point. On systems with shared caches, tasks using cache oblivious algorithms have a clear advantage over statically cache optimized tasks in that they degrade more gracefully as soon as other CPUs also use the shared cache. However, they neither allow to predict the performance loss of individual tasks, when different cache oblivious algorithms use a shared cache simultaneously, nor do they solve the handling of shared caches within parallel applications. For the latter, one can still gain advantages by letting tasks work on overlapping data sets, so there is less competition for the cache.

2.2 Resource Management

If architecture-specific optimizations as described in the previous section cannot be done – for instance, because the final application mix was not known at design

time or some piece of software cannot be recompiled for a different system – the operating system (or runtime library) can try to manage tasks, so that the usage of system resources is optimized. This often means avoiding situations where multiple tasks want to use the same resource at the same time. This resource contention usually results in a decreased performance for at least one task. For many internal system resources, where access is managed automatically by the hardware itself, simultaneous access causes short stalls in the execution of tasks. Examples include shared caches (where even the contents are managed in hardware), execution units in a SMT system, memory bandwidth, and others. These stalls cannot be masked by context switches, as it is customary for instance for I/O devices in an operating system. Thus, the time spend stalling is effectively lost. (Paging of main memory has a special role as it is one of the few cases where the operating system actively manages access to system resources and is able to mitigate the overall effect of stall with context switches to a certain degree.)

By controlling, which tasks are executed simultaneously and which are not, the operating system can reduce and sometimes even avoid the occurrence of such stalls. This has benefits for the system itself, which suddenly realizes more instructions per cycle, and for applications which exhibit lower response times on average. Today's multicore processors have a plethora of shared resources. They can be split into two broad categories: resources holding data, such as caches and main memory, which are limited by their capacity; and resources transporting data, which are mainly defined by their bandwidth (and their latency, though it is not important for the discussion here).

2.2.1 Improved Data Sharing

Shared memory, whether it is system memory or some cache, is limited. The more tasks are running, the higher is the combined demand. When the combined demand exceeds the available memory, the memory is effectively time-shared by repeatedly replacing cache lines or memory pages. This might result in a substantially decreased performance, because the hardware (for cache) or the operating system (for system memory) permanently has to fetch some piece of memory. This effect is called *thrashing* [29].

Memory Pressure

For system memory, thrashing means that a task – as soon as it is executed – runs into a page fault and requires a page to be swapped in. This puts stress on the I/O subsystem, which becomes the bottleneck and limits performance. The traditional solution is to reduce the number of applications, which reside in memory simultaneously, so that their combined working sets still fit into memory [29]. This approach can also be used with coscheduling as the ideas are mostly orthogonal [30].

However, carefully applied coscheduling reduces the need for such explicit two-level scheduling schemes in parallel systems. Firstly, coscheduling by itself when applied to parallel applications reduces the multiprogramming degree naturally: at any point in time there are only a few parallel applications running simultaneously. This reduces the combined working set across all CPUs in a system compared to uncoordinated scheduling. Secondly, with some coscheduling-aware paging policies the paging activities no longer influence the running applications negatively [31]. Finally, the control of the operating system can be used to prefetch the working set of the applications that are going to be executed next, so that all necessary swapping can be handled in the background [32]. The drawback here is that this style of coscheduling requires larger time-slices than those typically used today, though it is not worse than those of two-level schemes.

Cache Pressure

For shared caches in a multicore processor thrashing can make a cache ineffective: in the worst case every cache look-up fails and the next level of cache or the system memory must be consulted. Similar to system memory, coscheduling can be used to select tasks to execute simultaneously, so that their combined working set gets the most benefit from the shared cache. However, for caches it is more complicated to derive working set sizes because of the transparent management of caches. Additionally, other architectural details have to be taken into account – cache associativity for instance.

Still, with enough information it is possible to predict how a particular combination of tasks will perform when coscheduled on CPUs with a shared cache. To gather this information, tasks are executed in more or less controlled situations in order to generate profiles of individual tasks (e. g., [33]) or combinations of tasks (e. g., [34]). Once this data is available, appropriate groups can be formed. Most approaches, however, do not consider cache contents directly. Instead, they simply select combinations that reduce the overall number of cache misses as this is an easily retrieved measurement. Approaches, which focus on reducing the amount of data transferred, are discussed in the next section.

One example that directly targets cache contents is the work by Tam et al. [35]: they use performance counters to detect whether cache misses were resolved by remote caches. If so, tasks accessing the same data are not executing close together and coscheduled groups are modified to rectify that.

2.2.2 Reduced Resource Contention

For resources with access directly managed by the hardware itself, simultaneous accesses by multiple processor cores lead to stalls in the execution of tasks. These resources range from instruction fetching and execution unit allocation in SMT systems over accessing a shared cache to accessing off-chip resources, e. g.,

memory or I/O-devices. In most cases, access to hardware is arbitrated in a fair manner, giving each CPU an equal share of access opportunity. (One exception is, for instance, the SMT realization in POWER processors, where hardware threads can be prioritized [36].)

For devices or links that can be saturated by a single CPU, this leads to a worst case effect of losing any advantage of having multiple CPUs. The only way to surly avoid this kind of resource contention, is to have at most one task executing which accesses a particular resource. In most cases, this is not a sensible option as tasks are usually composed of varying mixes of resource accesses, which naturally overlap.

Execution Unit Contention

On systems with shared execution units, such as SMT systems where all execution units are shared or other systems where just some execution units are shared (the FPU for instance), it is beneficial to execute tasks simultaneously, that do not use the same shared execution units at the same time. Formulated differently: simultaneously executed tasks should complement each others' resource usage. This way, stalls while waiting for execution units (or other resources related to instruction execution) to become available are minimized.

The information that is necessary to make the required decision can usually be gathered from performance counters directly – either by observing and assessing combinations of tasks selecting only good sets to coschedule after a sample phase [37], or by assessing metrics of individual tasks and deciding on which to combine [38] Here, it works well to coschedule integer intensive with floating point intensive applications; or compute intensive and memory intensive applications: while the former are able to utilize most execution units at once when highly optimized, the latter make use of them only every few cycles and wait for data from cache or memory most of the time. However, as soon as cache and memory accesses are considered, there is no longer a clear distinction between this and the next use case.

Memory and Cache Bandwidth Contention

If CPUs share the same link to their cache or memory, coscheduling can be used to reduce contention on these links by selecting appropriate combinations of tasks to execute simultaneously on CPUs sharing certain links. Zhuravlev et al. [39] give an overview over different techniques achieving this.

Most techniques qualify the memory intensiveness of tasks by utilizing cache miss information in some way, which roughly corresponds to the amount of data transferred between two layers of the memory hierarchy. Then, some memory intensive tasks are coscheduled with some less memory intensive tasks, so that the available memory bandwidth is hopefully well distributed among the memory intensive tasks. Ideally, the combined demand does not even exceed the available bandwidth. There is also a slight overlap with the cache pressure use

case, because a combination of tasks might exhibit an even higher demand of bandwidth due to thrashing at an intermediate shared cache.

2.2.3 Physical Operation

In addition to the actual physical processor resources, there is also the less tangible resource energy, which has become a problem in recent years. The more power a system consumes, the more heat must be dissipated, so that the system is kept in an operational state.

Temperature Balancing

When the cooling budget in a computer system is limited, special care has to be taken, so that the system does not reach unhealthy temperature levels. With respect to processors, traditional approaches (see [40] for a survey) include reductions in operating voltage and frequency as well as the forcibly insertion of idle cycles. In a partly loaded multicore system, it is also possible to shift the load around, so that all processor cores are equally heated.

Going further, one can exploit that every task has its own energy characteristic: highly optimized tasks generally utilize execution units better; thus, they consume more energy and generate more heat than less optimized tasks. Considering a single CPU, it makes sense to interleave execution of such *hot* and *cold* tasks. On multicore systems, coscheduling provides the natural extension of this line of thought: by ensuring that hot and cold tasks are executed simultaneously on close CPUs (in addition to interleaving hot and cold tasks on individual CPUs), the overall heat development can be lowered (e. g., [41]). This delays or even avoids the point where additional mechanisms to address temperature hotspots have to be employed.

Power Capping

Even when cooling is not a problem, the power consumption itself might still be and the OS or hardware has to ensure that certain power levels are not exceeded by limiting the achievable frequency/voltage. Here, the operating system can exploit the individual energy characteristics of tasks and avoid to coschedule energy-hungry tasks (or enforce coscheduling of more and less energy-intensive tasks). This way, it may be possible to stay within a given power envelope at a higher frequency than it would be with an uncoordinated schedule.

Note, that it does not matter whether power consumption is capped in software (e. g., via ACPI P-states) or in hardware (e. g., via the *Running Average Power Limit* (RAPL) feature available in recent Intel Processors) – or whether the frequency/voltage is adjustable at a per-core or only at a per-socket level. In all cases, coscheduling enables the operating system to avoid peaks in the power consumption if the workload is varied enough.

Energy Contention

If the distribution of the energy budget is managed by the processor itself (i. e., the processor supports *Intel Turbo Boost Technology*, *AMD Turbo Core Technology*, or a similar technique), the processor automatically puts most of the energy budget to good use, e. g., by transparently increasing the frequency of the remaining cores when some cores transition into a sleep state or by transparently decreasing the frequency when a core executes particularly demanding instructions. However, due to this dynamically shared energy budget, the execution of a task on one core now affects the performance of tasks executed on other cores, slowing them down by contending for the shared resource energy. Coscheduling gives the operating system the ability to regain control about which tasks are allowed to be influenced by other tasks and which tasks should execute at peak performance. This use case is explored in detail later in Chapter 10.

2.3 System Design

In this section, coscheduling use cases are presented, which influence either the design of the operating system itself or extend or simplify the user/kernel-interface.

2.3.1 Application Management

Traditionally, coscheduling was the answer to the question how multiple parallel applications – where tasks heavily profit from synchronous execution – can be executed without having to resort to batch processing [1]. And while the management of such parallel applications is certainly one aspect, it is certainly not the only one since today's parallel applications take many different forms and shapes.

Parallel Application Management

Coscheduling allows a simple elevation of parallel applications to first class citizens. Instead of scheduling tasks, the operating system now schedules applications. CPU time can be managed at application-level instead of task-level, accounting for the fact that some applications are more parallel than others. This stops the selfish "more is better" mentality of some applications, which spawn tasks despite diminishing returns. With adequate accounting, an application creating tasks beyond its optimal number will only hurt itself.

Also, today's parallel applications are often moldable or even malleable as defined by Feitelson and Rudolph [42], giving the operating system a more active role in the management: for moldable applications, the operating system can select the degree of parallelism at startup (for example, MPI applications often belong into this category), while malleable applications can even be reconfigured

at runtime. Compared to scheduling approaches, which purely partition a system in space, coscheduling adds partitioning in time as a new dimension to the solution space. This added flexibility can be utilized in different ways (e. g., [43–45], Chapter 9). In particular, coscheduling has the ability to reduce the management overhead and allows easy handling of workloads with non-malleable applications in the mix. A more in-depth discussion of this use case can be found in Chapter 9.

Additionally, Chapter 11 dives into a more specific example, where algorithms to avoid resource contention are now applied to parallel applications instead of individual tasks.

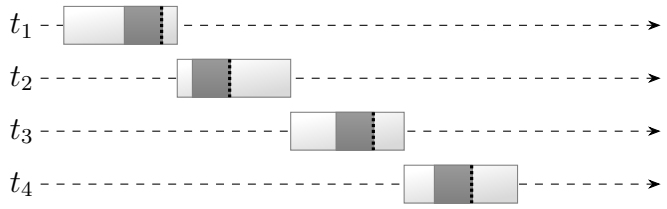
Concurrency Management

Bridging the areas of application and system design, coscheduling can be used to control the number of related tasks that are executed simultaneously at any point in time. If the execution of related tasks is not coordinated, the expected number of simultaneously executing related tasks is related to the overall number of runnable tasks in the system: the more tasks there are, the higher is the probability that related tasks are not executed simultaneously. That is, the expected degree of concurrency within a group of related tasks decreases with more tasks and increases with less. Coscheduling allows to control this degree explicitly and independently from the actual number of runnable tasks within a group.

Limiting the number of simultaneously executing tasks in a group (without limiting their overall number) can be used to lower contention for shared data structures. When the overall load is large enough, the machine can still be operated at full utilization. If these shared data structures are accessed with blocking algorithms, tasks will block less often – though lock holder preemption might need to be handled separately. For lock-free algorithms, limiting the number of simultaneous executing tasks results in less retries. Obstruction-free algorithms will abort less often and can even be brought out of a livelock, if time slices are long enough. This is of particular interest for transactional memory systems, as they fall mostly into the obstruction-free category [8, 46].

Figure 2.8 illustrates such a case with four task optimistically trying to change a shared data structure. When a task recognizes that another task has modified the data structure in the meantime, it simply tries again. In a system without any scheduling coordination across CPUs, it is up to chance whether we will see the best case (Fig. 2.8a), the worst case (Fig. 2.8b), or – more likely – something in between. Coscheduling can enforce either situation; the best case by forbidding simultaneous execution. However, if there is not enough load, it might be better to still allow two of those tasks to execute simultaneously to avoid an idle system.

The other way around, simultaneously executing as many related tasks as possible is the normal mode of operation for coscheduling. This can also be used in a slightly modified variant, where not all tasks have to execute simultaneously



(a) Best case (at least nearly, some intertwining is still possible).

(b) Worst case.

Figure 2.8: Four tasks optimistically updating a shared data structure; retrying when another task was faster. Without coscheduling, any schedule is possible. Explicitly limiting the degree of concurrency with coscheduling enforces the best case.

but at least a certain number of them (when there are more tasks than that), giving the OS scheduler a little bit more flexibility. A high degree of concurrency is useful for certain classes of parallel algorithms that actually work more efficient with more simultaneously running tasks, such as software combining and elimination [8].

This use case is covered in more detail in Chapter 11.

Affinity Management

Coscheduling also removes many of the reasons why applications manage task affinities by themselves. Typically, the topology of the system is somehow important for an application, for example to realize some architecture-specific optimizations. Another reason for direct affinity management is to avoid load balancing artifacts of the operating system scheduler, which might even occur in dedicated scenarios: if a set of tasks becomes runnable, it is not necessarily spread out among the CPUs. The tasks must at first generate load, so that the load balancer will consider them. Thus, a parallel application might not realize its full potential until the balancer did its job.

Coscheduling naturally solves the latter: runnable tasks are always executed simultaneously; they are spread out by definition. For the former, most application actually do not require a specific task to CPU placement, but only a placement of tasks relative to each other. Take a VM as an example, to which the host's SMT capability has been passed through. For a bunch of SMT siblings it does not matter on which processor core they are executed, as long as

all siblings are on the same core. This is something that a coscheduler must support anyway in order to realize many of the other coscheduling use cases.

Not mapping tasks explicitly at application level gives the scheduler more freedom to shuffle things around for optimal performance. This use case is explored later in Chapter 11.

2.3.2 Security

The guarantee of simultaneous execution (or non-simultaneous execution, depending on the point of view) provided by coscheduling can be used to thwart a subset of side-channel attacks.

Side-Channel Elimination

Side-channels pose a risk on multiprogrammed systems running untrusted code, as said code may attempt to extract secrets by intelligently monitoring the system and its running applications. Ge et al. [47] survey various attack methods and countermeasures.

With the introduction of multiprocessing, malicious code is no longer limited to time-sliced execution with victim code: it is now able to run simultaneously with victim code – enabling new methods of attack. The effectiveness of such attacks depends on different factors; the rule of thumb is, that the more resources are shared between CPUs, the more fine-grained observations can be made, increasing the chance of finding a successful attack. Countermeasures are as varied as the attacks themselves. For example, a typical countermeasure on the application-side is to ensure that the observable behavior does not differ for varied inputs. At system level, one possibility is to make accurate observations harder by injecting artificial noise; another is to disable SMT, depriving an attacker of one of the more accurate observation options (see, e. g., [48] for an approach that combines aspects of different countermeasures).

With coscheduling, it is possible to dynamically disallow simultaneous execution of untrusted code with sensitive code. Hence, sensitive code can run in isolation – on a core, on a processor, or within the system, depending on the use case – without the permanent loss of parallel computations, reducing the available side-channels.

A concrete example is the mitigation of the VM-based variant of the Foreshadow/L1 terminal fault (L1TF) vulnerability on SMT-enabled Intel systems: if the host utilizes extended page tables (EPTs), a malicious VM can read any data that is currently held within the L1 cache [49, 50]. If the user does not trust their VMs, they have basically two mitigation options: Either disable the use of EPTs and fall back to shadow paging, taking a severe performance hit especially when mitigations against Meltdown are in place within the VM [51, 52]. Or ensure that the L1 cache is free of secrets, which is achieved by a combination of L1 cache flushing when entering a VM and ensuring that the other

SMT-sibling cannot execute anything, that might load sensitive data into the cache. Coscheduling at core-level of tasks executing guest-code of the same VM achieves this objective without having to disable SMT.

2.3.3 Device Management

One function of an operating system is the management of devices. Usually, the operating system provides access to physical devices to applications only through some layers of abstraction, e. g., files for storage on a disk, sockets for communication over network cards, or virtual devices instead of real ones when operating as a hypervisor. This typically requires request queues and buffers within the operating system to successfully share such devices between multiple applications.

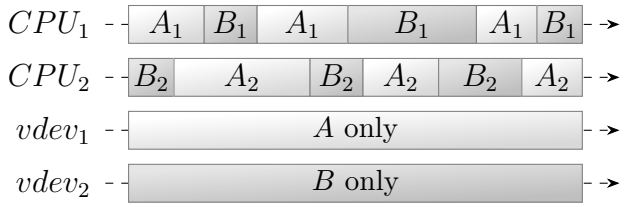
These abstraction layers increase the memory footprint of the operating system and also induce some computational overhead. And even then, it is impossible in a multiprogrammed system to use some of the more specialized functionality at application level, such as dynamic voltage/frequency scaling or networks for collective operations. Hence, there are some attempts to reduce this overhead again and to make special functions available to applications – particularly within the virtualization and HPC areas.

Exclusive Device Access

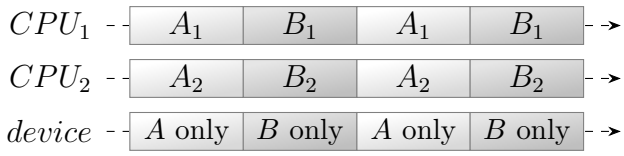
The basic idea is to reduce the number of applications that use a particular device to one, i. e., to artificially "unshare" a device. With hardware support, this can be done by moving the abstraction layer into the hardware itself and provide multiple views of the same device (e. g., via single root I/O virtualization (SR-IOV) as demonstrated in, e. g., [53]). This basically results in one exclusively owned device per application. Without hardware support (or if the hardware cannot provide enough virtual devices), coscheduling can be used to reduce the number of simultaneously executing applications, so that at most one application accesses a particular device at a time. The device state is then included in every context switch of the application. Both variants are depicted in Fig. 2.9. Due to their different realization, both variants have their own use cases which overlap only partially.

For the coscheduling variant, the device in question must be free from external influences as it is practically non-existent when the application is not running. Originally, this was used to optimize the communication layer within a cluster with multiple parallel applications [54,55], where applications are allowed to directly access device buffers and use advanced functions of the connection network. This will become interesting again with the upcoming manycore processors and their fast, specialized on-chip communication networks (e. g., [56, 57]).

Switching device state can be more or less costly, which has to be included in the cost calculation. For communication devices, for example, messages that are in transit during a context switch require special attention.



(a) Exclusive device access via hardware I/O virtualization. The same device presents multiple views of itself. Each view is assigned exclusively to an application.



(b) Exclusive device access via coscheduling. An application can access the device whenever it wants, coscheduling ensures that at most one application can access a device at a time.

Figure 2.9: Two different techniques to "unshare" a normally shared device in a multicore environment.

Expose System Functionality

The operating system cannot give applications control over functionality, that directly influences the performance of the system or might even compromise its integrity, as this would create a conflict of interests between multiple applications. However, if coscheduling is used to make sure that tasks from the same application are executed on all affected CPUs simultaneously, it is possible to expose (moderated) control over such functionality to applications.

For example, control over dynamic voltage/frequency scaling (DVFS), certain cache and memory controller settings, or per-core/per-package performance counters can be exposed to applications. Similar to the previous section, their settings and current values have to be included in a context switch. This gives applications the ability to steer their part of the system towards their own optimization goal – at times when they actively use the system. Examples include a pro-active DVFS control instead of the typically reactive approach taken by operating systems [58], and – in case of VMs – it allows moving the control loop from the host into the guest, which can make more informed decisions about the desired energy/performance trade-off for its workload.

Special Architectures

From a hardware design point of view, coscheduling also opens the possibility to physically share devices which are typically realized per-CPU, such as the memory management unit or a virtually tagged cache. Coscheduling can guarantee that only tasks from the same application will access said device at any time.

This would create a middle ground between full-fledged SMP systems and single instruction/single data (SIMD) machines: while still able to execute multiple instructions on multiple data elements at any one point, these would need to belong to the same address space. Additionally, other hardware concepts such as partner cores [59] can be supported by a general purpose operating system.

2.4 Chapter Notes

The use cases presented in this chapter will be analyzed from a coscheduling perspective in Chapter 3: what are their particular requirements from a coscheduler, and how can a single scheduler support them all. Note, however, that while this thesis considers all described use cases at the theoretical level, it will not actually realize every single use case. This has for most cases already been done by other people as made evident by various references and does not have to be repeated. That said, some specific use cases will be covered in more detail in later chapters: Chapters 9 and 11 deal with system design, while Chapter 10 covers energy contention and also touches the topic of exposing shared frequency controls to applications.

The problem of energy contention was first published by myself in [60]. As far as I know, I am the first with this thesis to suggest coscheduling as a mean to improve response times of interactive applications and throughput in power-capped scenarios, or as an alternative mean to realize concurrency and affinity management.

Chapter 3

Coscheduling Model

Since their introduction in 1982 and 1990, the concepts *coscheduling* [1] and *gang scheduling* [43] have been used by various authors. Due to new use cases and newly identified aspects, however, many different interpretations of these concepts were created and their original meaning has been – while not lost – somewhat diluted. Additional confusion was created due to *coscheduling* sometimes being used as in the original definition from Ousterhout [1], and sometimes being used as it is realized (a bit differently) by the algorithms presented in the very same publication.

Today, coscheduling and gang scheduling are usually used as umbrella terms for techniques, that achieve some kind of simultaneous execution of certain tasks. In their original meaning, both are not versatile enough to cover every aspect that arises today. And while many techniques have been developed that realize some kind of coscheduling, there is still no terminology to adequately express coscheduler capabilities or requirements of use cases. To address this, a model for coscheduling is developed in this chapter. This model is able to capture the essence of coscheduling related aspects of a scheduler or a use case, which – for instance – allows to easily select an adequate scheduler for a certain use case or a set of use cases.

The chapter begins in Section 3.1, where the common vocabulary is established that is used in this thesis. Section 3.2 collects several coscheduling-related definitions and terms, that influenced the nomenclature used within the model. Then, the model itself is constructed by first analyzing coscheduling use cases and their requirements with respect to coordination in time (Section 3.3) and coordination in space (Section 3.4). These insights are then combined into the desired model in Section 3.5. Finally, Section 3.6 examines different levels of adherence to the model and their consequences for coschedulers and use cases.

3.1 Basic Terms

In the literature, terms like coscheduling are not the only ones with varying meanings. Even more confusion can arise around much simpler terms, such as

process, thread, or processor. Is a process something that can be executed, or is it just a container for threads? Is a processor a piece of hardware that is plugged onto a motherboard, or is a processor just able to execute a single strand of instruction? As this list goes on and on, this section will clear possible ambiguities of basic terms used in this thesis.

3.1.1 Software

On the software side, the basic unit is a *task*: a single strand of instructions that are to be executed within their own context. Contrary to the terms process or thread, the term task shall have no ingrained notion of the extend of its context. Throughout this work, tasks will often be executed simultaneously. There shall be no preconceptions about their relations. On the one end of the spectrum, they can operate closely within the same address space; on the other end, they can be completely unrelated.

A bunch of tasks, that work towards a specific goal and were developed in conjunction, form an *application*. The applications in this work are mostly parallel in nature – though the degree of teamwork will vary. Sometimes it also makes sense to consider certain unrelated tasks as a unit. As a generic term for any specific conglomeration of tasks, *task group* will be used.

The term *scheduling entity (SE)* will be used for something that is scheduled by the OS scheduler. As such, every task is also a scheduling entity. However, tasks are not the only SEs in this thesis: a group of SEs may be represented by a scheduling entity of its own. From the point of view of the OS scheduler, a scheduling entity is in one of three possible states: running, ready, or blocked. For a task, these states are defined as usual:

Running (task): The task is currently being executed on a CPU.

Ready (task): The task is ready to be executed, i. e., it is currently waiting for a CPU in the scheduler's runqueue.

Blocked (task): The task cannot be executed, currently. (Usually the task is waiting for something.)

While operating systems often distinguish further states, these three are universally existent and a further distinction is not necessary for this thesis.

For a scheduling entity that represents a group of SEs, the states have to be defined differently, though the essence is preserved:

Running (group): The state of at least one represented SE is running.

Ready (group): No SE is running, but a transition to running is possible.

Blocked (group): The group is neither running nor ready.

If a SE is either ready or running, it is also referred to as *runnable*.

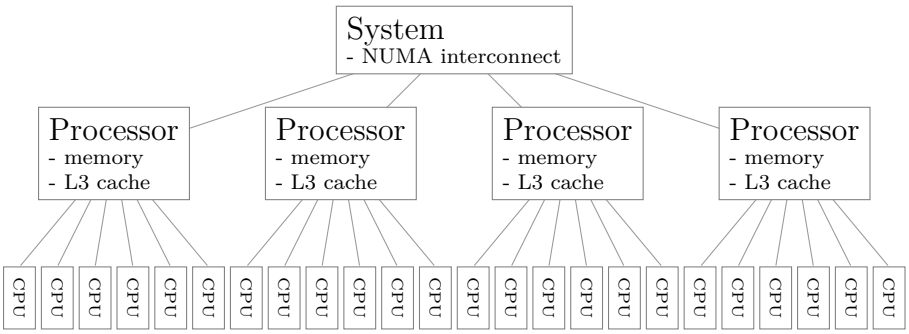


Figure 3.1: The system used in Chapter 9 represented as a tree of topological units. Every CPU comes with its own set of execution units and an L1/L2 cache.

3.1.2 Hardware

One the hardware side, multicore architectures have become complex pieces of technology. In this thesis, a *CPU* is the basic unit in a multicore system; it is able to execute just a single strand of instructions, i. e., a task. In a SMT system, a *processor core* consists of multiple CPUs. Multiple processor cores make up a *processor*. Multiple processors may occupy the same *non-uniform memory access (NUMA) domain*, i. e., they have similar performance characteristics when accessing memory. The whole *system*, finally, may consist of multiple NUMA domains.

Most of today's multicore architectures can be described with this, and it is fine when talking about actual systems. Conceptually, however, this thesis abstracts from these concrete hardware components and considers multicore systems to be composed of multiple *topological units (TUs)*. These TUs are arranged in a tree according to the system topology. The leaf TUs are CPUs, while non-leaf TUs usually represent interconnection networks connecting their children. Additionally, every TU may have some resources associated with it: for example execution units, cache or memory. The topology of interconnection networks themselves does not matter for most parts of the thesis. They can be bus-like, which is often the case within today's processors, or structured as the interconnection networks between NUMA domains.

An example topology tree is shown in Fig. 3.1, which shows one of the evaluation systems that is used later on. The structure of this specific topology tree is relatively simple. In a SMT system, CPUs and processor cores would be on different levels – with execution units attached to cores instead of CPUs. Also, it is possible to adequately represent, for instance, AMD's Bulldozer microarchitecture [23] with execution units at CPU and at processor core level. Complex interconnection networks, for example the NUMA interconnect in the SGI UV architecture [61], would be similarly decomposed and represented as multiple levels within the topology tree.

3.1.3 Parallelism

When considering parallel applications or task groups in general, a characterizing property from the point of view of the OS is the number of runnable tasks in a group and whether that varies over time. The number of runnable tasks is referred to as *degree of parallelism (DOP)*. Plotting the DOP over time results in a *parallelism profile* for that task group. Depending on the system and its load, it is usually not possible to always execute all runnable tasks. The number of running tasks is referred to *degree of concurrency (DOC)*. The DOC over time results in a *concurrency profile*. That is, this thesis uses parallelism to refer to a feature of an algorithm, while concurrency refers to the realization of parallelism by the operating system.

Following [42], this thesis distinguishes four classes of task groups:

Rigid: A rigid task group has a fixed degree of parallelism.

Evolving: An evolving task group changes its DOP during execution.

Moldable: A moldable task group is rigid, with the exception that the fixed DOP is supplied by the OS.

Malleable: A malleable task group is evolving, with the exception that changes in the DOP are triggered by the OS.

So, in order for the OS to be able to control the DOP of a task group, the task group must be malleable. The DOC, on the other hand, can always be controlled by the operating system.

3.2 Cherry-picking Coscheduling Terms

This section describes coscheduling related terms and definitions that influenced the coscheduling model developed afterwards. Every concept includes a bit of background and how the original idea is reflected in the new coscheduling model. For the sake of simplicity, a term is always attributed to the first author of the publication, where it first appeared.

3.2.1 Ousterhout's Coscheduling

Ousterhout's work [1] from 1982 defines the term *coscheduling* and presents three algorithms to achieve coscheduling. It is regarded today as the key work in that area. This and his earlier work coauthored with Scelza and Sindhu [62] together form a summary of his Ph. D. thesis [63] of which a revised version was published as a book [64] in 1981. The thesis is about the design of a distributed operating system called Medusa.

Ousterhout's coscheduling definition revolves around groups of closely cooperating tasks, which he calls *task forces*:

> "A task force is coscheduled if all of its runnable processes are executing simultaneously on different processors. Each of the processes in that task force is also said to be coscheduled." [1]

Task forces are considered to be cooperating activities with fine-grained interactions based on locks. The phenomenon of individual tasks repeatedly blocking on locks, because of a not coscheduled task force, is called *process thrashing* by Ousterhout.

Ousterhout also coined the term *fragmented execution*, which refers to cases when at least one task of a task force is executed without being coscheduled according to the definition above. Another notable term is the *Ousterhout matrix* as it is called by other authors today. In one of his algorithms, Ousterhout used a matrix to describe the schedule. The matrix has one column per CPU, while the rows represent different task slots. The idea is to fill this matrix with task forces and ensure coscheduling of rows of that matrix by using a synchronized round-robin scheduler. For a more thorough description see Section 4.1.

This work uses the term coscheduling in Ousterhout's original meaning when being formal: the model defines a coscheduling constraint. In less formal situations, coscheduling is simply used as a synonym for simultaneous execution. Additionally, this work uses the term *gang*, which is conceptually equivalent to a coscheduled task force.

3.2.2 Feitelson's Gang Scheduling

Feitelson and Rudolph defined the term *gang scheduling* in their work [43] from 1990. From then on, both worked together and also independently with others on this topic on and off during the next 15 years. This resulted in a wide range of publications ranging from mostly analytical work to more practical applications of gang scheduling. The initial work on that topic defined gang scheduling as follows:

> "[Gang scheduling is defined] as the scheduling of a group of threads to run on a set of processors at the same time, on a one-to-one basis." [43]

A later survey [65] added the requirement for time slices and collective preemption to the definition of gang scheduling, which was only implied beforehand.

The one notable difference of gang scheduling to Ousterhout's coscheduling is that tasks do not relinquish CPUs during their time slice. Even in a blocked state, a task still occupies its CPU as long as its group is executed. In such a case its corresponding CPU will be idle. Ousterhout's coscheduling is less strict and principally allows a CPU to pick up some other task, while its actual task is blocked. Thus, gang scheduling prevents the cache from being polluted and promotes the idea of actually owning the hardware while a task group is executed.

This work will use the term *gang* roughly in this original meaning. Though, CPUs are allowed to pick up other work. A variant of a gang, which achieves the original meaning (and a bit more), is an *isolated gang* in the terminology of the model.

3.2.3 Arpaci-Dusseau's Implicit Coscheduling

Arpaci-Dusseau defined a coscheduling mechanism for clusters in 2001 that coschedules tasks not based on explicitly specified groups of tasks, but based on implicit information. This *implicit coscheduling* [66] is realized by an especially crafted local scheduling policy and an adaptive waiting scheme that monitors communication. Specifically, the described mechanism relies on tasks being woken on message arrival and their utilization of spin-blocking, where the waiting time is determined dynamically based on the ideal round-trip time and arrival rate of incoming messages. This way and together with a proportional-share scheduler, communicating tasks are automatically coscheduled until their time slices are used up.

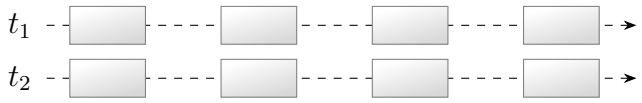
This work will use *implicit coscheduling* as a broader term for a family of techniques that achieve a desirable coscheduling without actively selecting tasks to be coscheduled. The opposite of implicit coscheduling will be called *explicit coscheduling*. These implicit techniques combine several seemingly unrelated methods, which realize the desired coscheduling as emergent behavior.

For example, another implicit technique was suggested by Merkel et al.. This Sorted Coscheduling [67] focuses on the reduction of resource contention. This is done by distributing tasks with equal resource demands evenly across CPUs and then using a special scheduling policy that executes tasks on a CPU in order of their expected resource demand. By alternating the execution order within every pair of CPUs and selecting appropriate time-slice lengths, tasks with contrary demands are coscheduled.

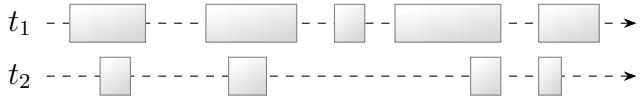
3.2.4 VMware's Relaxed Coscheduling

The scheduler in VMware's vSphere Hypervisor [9, 68] is capable of a form of coscheduling. Contrary to the coscheduling variant implemented in their earlier product, the current version supports what they call *relaxed coscheduling*. This relaxed coscheduling does not enforce simultaneous dispatching and preemption as Feitelson's gang scheduling. Thus, a fragmented execution is principally allowed. The possibly negative effects of fragmented execution are contained by limiting the allowed *skew*, i. e., by limiting the difference in vCPU progress.

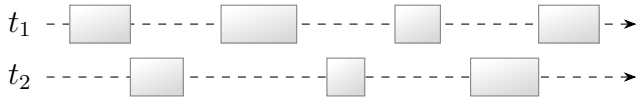
This thesis uses the term *relaxed* in situations, when a certain scheduling guarantee – such as simultaneous execution or a certain placement – is not enforced (or required) all the time. Instead, some (minor) deviations from a guarantee are allowed, which gives a coscheduler implementation more freedom. If applications do not require a *strict* adherence to a guarantee, they are expected to handle such a deviation somewhat gracefully.



(a) Classic two-sided coscheduling. Whenever at least one task of a group of runnable tasks is running, all tasks of said group are running.



(b) One-sided coscheduling. A task may only be executed, when a certain other task is running, but it does not have to.



(c) Anti-coscheduling. A task may not be executed, when a certain other task is currently executed.

Figure 3.2: Different interpretations of coscheduling. All three variants have merit.

3.3 Aspects of Time

Coscheduling is – in essence – the controlled simultaneous execution of tasks in a parallel system. That is, a coscheduler actively controls which subset of tasks is running at any point in time. This control can come in two flavors: tasks can be forced to execute simultaneously, or their simultaneous execution can be prevented. Traditionally, coscheduling is about the former variant: exploiting synergies of synchronous execution. But especially with resource contention in mind, the second variant can also be attractive: avoiding simultaneous execution of tasks that stress the same resources.

A particular point of simultaneous execution is, that it does not have to be symmetrical as described by Ousterhout [1] and Feitelson [43]. Just because a task A should always be executed while task B is running, it does not mean that A must always be executed when B is running. All three interpretations of coscheduling are depicted in Fig. 3.2. This leads to a definition of coscheduling, that is slightly different from all those definitions and interpretations previously published.

Definition 1 (Coscheduling)**.** A task is said to be coscheduled with another task, when it is only executed when this other task is also currently executed (cf. Fig. 3.2b).

This asymmetric definition allows to cover more use cases, specifically certain types of auxiliary tasks and an adaptive solution to handle lock holder

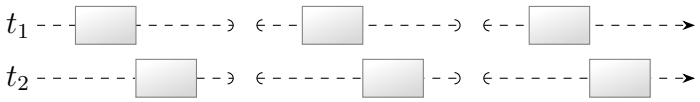


Figure 3.3: Not coscheduling. Enforcing an execution order of tasks – here t_2 is always executed directly after t_1 – is not in the scope of coscheduling.

preemption (both covered later in Chapter 11). What even this extended definition of coscheduling does not – and should not – cover, is the execution order of tasks. That is, it is impossible to specify constraints like "task A is always executed directly after task B" (cf. Fig. 3.3). While it would make sense in some cases to use such a constraint (e. g., because tasks A and B share data and doing so would increase the cache hit rate), this addresses a problem that is also present on uniprocessor systems. And this thesis considers coscheduling to be a multiprocessing problem, that strictly addresses the topic of simultaneousness.

When not considering just two tasks, but potentially larger groups of tasks, only the traditional symmetric coscheduling behavior makes sense: all tasks should be executed simultaneously. However, with groups of tasks it is now necessary to specify group behavior when not all tasks within the group are runnable. Both, Ousterhout and Feitelson, agree that coscheduled tasks should be able to block individually instead of one blocking task causing the whole group to get blocked. This leads to the following definition of a gang.

Definition 2 (Gang)**.** A task group is said to be a gang, when all runnable tasks are coscheduled with each other, i. e., all runnable tasks are always executed simultaneously (cf. Fig. 3.2a).

This means in particular, that a gang is always able to realize its full parallel potential and that its concurrency profile is identical to its parallelism profile. However, neither Ousterhout nor Feitelson (nor anyone else) gives a definition for coscheduled groups of tasks, that can be used throughout all use cases. In particular, it is necessary to break the coupling of the number of coscheduled tasks to the number of runnable tasks in a group for many of the multicore-specific use cases. Removing this coupling yields a very general definition, which basically says that simultaneous execution is done on purpose and not by accident.

Definition 3 (Scheduled task group)**.** A task group is said to be scheduled, when the degree of concurrency of the task group is actively controlled by the scheduler.

This definition in itself is too broad to be of much use. For that, the control exerted by the scheduler needs to be defined. This is done in the form of freely combinable constraints that describe the desired concurrency profile in more detail. To regain the behavior of a gang as defined above, the degree of concurrency needs to be maximized, basically reintroducing the just broken decoupling.

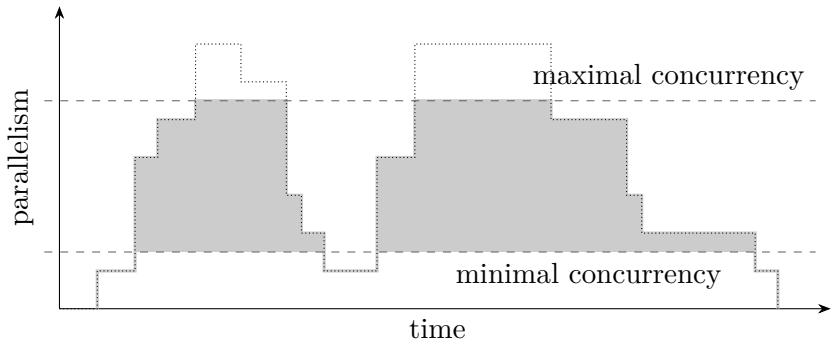


Figure 3.4: Effects of concurrency constraints on a scheduled task group. The scheduler can freely select the degree of concurrency within grey area.

Definition 4 (Coscheduling constraint)**.** The degree of concurrency within a task group is always equal to the number of runnable tasks, whenever a task is executed.

This constraint is needed mainly for situations that include active waiting or other forms of synchronization, where a synchronous execution of tasks is indicated.

In situations, where simultaneous execution of all runnable tasks is not strictly required, it may still be necessary to restrict the degree of concurrency, so that it either does not go above a certain level or below. This – in its pure form – is mostly needed for concurrency control use cases. Additionally, it is implicitly included in many of the placements constraints that are discussed in the next section.

Definition 5 (Minimal concurrency constraint)**.** The degree of concurrency within a scheduled task group never goes below a certain value, while it is executed and enough runnable tasks are available. When there are not enough runnable tasks, the DOC is kept equal to the DOP, whenever a task is executed.

Definition 6 (Maximal concurrency constraint)**.** The degree of concurrency within a scheduled task group never goes above a certain value. If more runnable tasks are available, care is taken to always execute only a subset of the available tasks simultaneously.

These constraints gives the scheduler a bit more freedom than the coscheduling constraint. Their effects are illustrated in Fig. 3.4, which shows a parallelism profile with allowed degrees of concurrency shown in grey. Please note, that a reduction of concurrency would result in a prolonged execution in reality.

With the minimal concurrency constraint, a group is still scheduled when there are not enough runnable tasks to reach the minimal degree of concurrency. The option to cease execution of a task group, when the number of runnable tasks drops below a limit, also exists.

Definition 7 (Minimal parallelism constraint)**.** If the number of runnable tasks within a scheduled task group goes below a certain value, no task of the group is executed.

This constraint is mainly needed for applications with hard demands on synchronous execution so that – for instance – in the event of one task running into a page fault, the others stop executing immediately. However, it must be applied carefully in order not to accidentally stop execution of a group, where only one of the remaining runnable tasks would create or unblock other tasks within the same group. Usually, this constraint has to be driven dynamically to account for voluntary blocking and an allowed skew in progress caused by involuntary blocking. A constraint to cease execution when there are too much runnable tasks does not make any sense (except maybe to detect erroneous application behavior), as runnable tasks usually do not disappear without being executed.

3.4 Aspects of Space

When a scheduled task group does not span the whole system, the question of *where* within the system it should be executed must be answered. Here, each scheduled task group may have its own idea of a supposedly good mapping. Traditionally, a task group was equivalent to a parallel application. And especially on clusters the best practice usually is: execute tasks of a task group close together, so that communication delays across the interconnection network are minimized. With today's NUMA systems and also cache architectures, this style of placement is also relevant for individual computer systems. This is again expressed as a constraint that can be attached to a scheduled task group.

Definition 8 (Close placement constraint)**.** A task group is mapped to a topological unit as deep as possible within the topology tree, that can still accommodate the DOC of the task group. Within the selected topological unit, CPUs are selected so that distances are minimized.

This constraint causes a task group to be mapped to a set of CPUs that share as many resources as possible, which should provide reasonably low communication costs. The support of a variable DOC makes this constraint particularly useful for parallel applications, where the degree of parallelism is not tied to certain architectural properties. Possible placements of task groups of different degrees of concurrency are exemplified in Fig. 3.5.

Use cases that address resource contention or contain architecture-specific optimizations, need more specific constraints. Namely, that a certain resource is shared between CPUs used by a task group – or the direct opposite, that a certain resource is explicitly not shared. Possible resources are: execution units, different levels of cache, memory, communication networks, etc. Additionally,

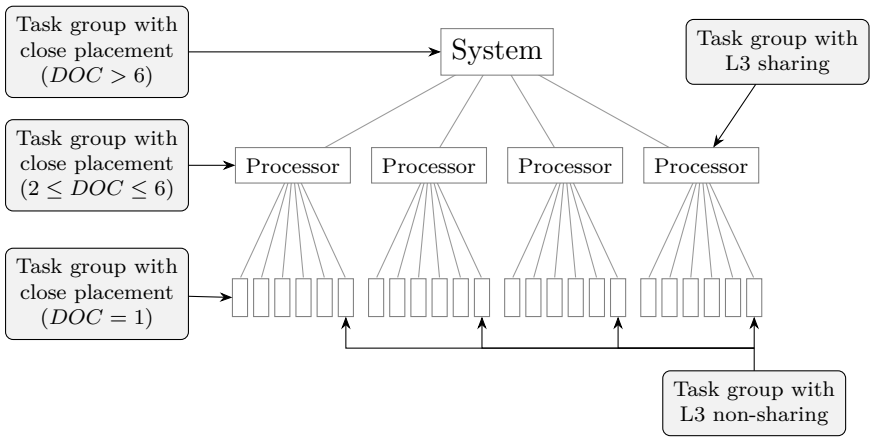


Figure 3.5: Picking up the topology tree from Fig. 3.1, this shows possible mappings of task groups with certain placement constraints within the system.

there are less tangible resources like frequency controls, shared power budgets, cooling capacities, and possibly more.

Definition 9 (Resource sharing constraint)**.** All tasks of a task group are mapped to a topological unit (or below it), to which the resource in question is attached.

Definition 10 (Resoure non-sharing constraint)**.** Tasks of a task group are mapped to a most one CPU within each topological unit that has its own instance of the resource in question.

Both constraints limit the number of CPUs allocatable to a task group. Thus, there is an implicit maximal concurrency constraint. In the example given in Fig. 3.5, the DOC may be at most six, when L3 cache should be shared, and four when each task should be able to access a different L3 cache, so that there is no interference of other tasks of the same group. These constraints differ from the close placement constraint, in that they are tied to topological units at a certain level of the topology tree, while the close placement constraint adapts to the task group possibly encompassing the whole system.

Some task groups are able to fully utilize resources of a topological unit, but without using all of the CPUs. Such cases need an additional guarantee that interference within the topological unit by tasks of other groups will not happen.

Definition 11 (Isolation constraint)**.** When a scheduled task group is executed, no other tasks are executed simultaneously that might interfere by accessing the isolated resource.

A coscheduled, isolated task group resembles Feitelson's definition of a gang, that keeps unused CPUs idle. The only difference is that the number of CPUs is primarily determined by the hardware instead of the overall number of tasks

within a task group. Still, both decouple the number of assigned CPUs from the number of running tasks.

Finally, it is possible to define the opposite of the close placement constraint. Though, the goal is not to maximize communication costs, rather tasks should share as few resources with each other as possible.

Definition 12 (Independent placement constraint)**.** Tasks within a task group are spread throughout the system, so that the potentially available resources are maximized.

In some sense, this is a variant of the non-sharing constraint, that does not specify an explicit resource. Thus, it is able to grow and shrink with the concurrency of its task group, similar to the close placement constraint. This constraint basically resembles the default scheduling policy of most operating system schedulers that have an understanding of today's multicore architectures: using first one CPU per processor before utilizing the others – and then using just one CPU per core, while enough CPUs are available.

3.5 Coscheduled Sets

With a scheduled task group and attached constraints, it is now possible to describe the scheduling requirements of most use cases of Chapter 2 adequately. But especially for nested use cases, the concept is not yet flexible enough. While the close and independent placement constraints work with arbitrarily degrees of concurrency, they lack the more fine-grained expressiveness of the resource-specific constraints. These, on the other hand, impose concurrency limits on scheduled task groups. It is, for instance, impossible to define a scheduled task group for a VM with a passed through system architecture; or to express nested parallelism, where the outer level utilizes NUMA domains and the inner level CPUs of a multicore processor. Also, it is impossible for an operating system to optimize the mapping of a parallel application that already requested to be coscheduled. To facilitate these and other scenarios, scheduled task groups are generalized into *coscheduled sets*, which achieve the desired level of expressiveness.

Definition 13 (Coscheduled Set)**.** A coscheduled set is the combination of a group of one or more scheduling entities and a set of scheduling constraints. A coscheduled set is a scheduling entity itself. A nested coscheduled set may not contain conflicting constraints.

Scheduling constraints on groups of SEs are defined similarly to the constraints on scheduled task groups known from the previous two sections. These constraints just operate on SEs instead of tasks, and the definitions for DOC and DOP are based on the number of running or runnable SEs, respectively.

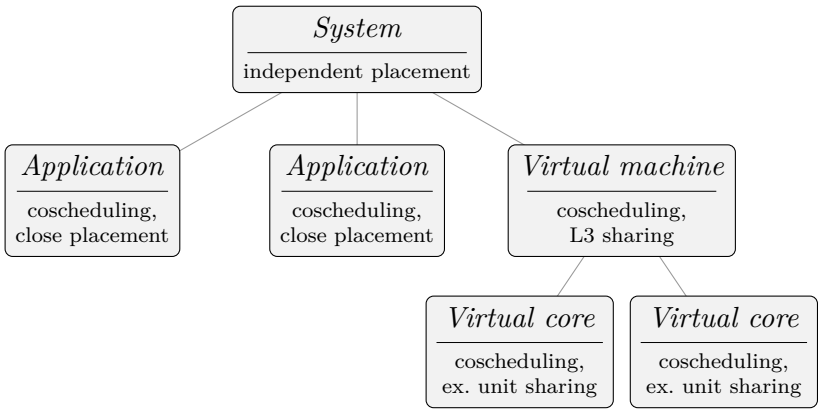
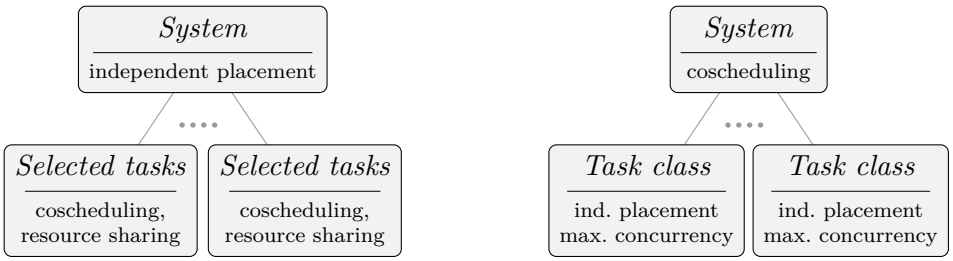


Figure 3.6: Exemplary setup of coscheduled sets for different kinds of parallel applications: two simple parallel applications and a multicore VM that is able to effectively utilize SMT.

With these definitions in place, it is now possible to describe all use cases presented in Chapter 2. Use cases in the area of application and system design usually have a straight-forward translation into coscheduled sets: For each parallel application, a (possibly nested) coscheduled set is created. This set then often exists for the lifetime of the application. Depending on the structure of a program, coscheduled sets can also be created or modified on demand for certain execution phases.

For example, a simple parallel application could be described by a coscheduled set with a coscheduling constraint and a close placement constraint. A system is able to handle multiple of these applications by adding their sets to a coscheduled set with an independent placement constraint. A more complex parallel application, for instance a VM which takes advantage of the multicore and SMT architecture of its host, would consist of multiple levels in itself: SMT siblings should be executed simultaneously and they should be placed within the same core of the host. Thus, for each core within the VM there is a coscheduled set with a coscheduling constraint and an execution unit resource sharing constraint in which an appropriate amount of vCPUs is placed. (An optional execution unit isolation constraint keeps performance within a partly loaded VM similar to a non-virtualized at the cost of idle computational resources in the host.) These coscheduled sets, in turn, are placed in a coscheduled set with a coscheduling constraint and an L3 cache resource sharing constraint, so that the cores of the VM are executed simultaneously and placed within the same processor within the host. Both types of parallel applications are illustrated in Fig. 3.6. Of course, many different setups are possible to support different use cases.

For resource management use cases, which mostly consider behavior of individual tasks in order to executing good combinations of them simultaneously, the translation into coscheduled sets is less straight-forward. Depending on the



(a) Independent execution of possibly many coscheduled sets with specifically tailored good task combinations.

(b) Coscheduling of a few task classes. Any possibly resulting combination of tasks is good.

Figure 3.7: Different ways to handle resource management use cases.

actual use case and the executed applications, behavior of individual tasks may change more or less often – in addition to dynamic creation and termination of tasks. Basically, there exist two ways to derive coscheduled sets for these scenarios. The first possibility is to organize tasks into multiple flat coscheduled sets, with each set being optimized for the considered resource. Thus, each coscheduled set has a coscheduling constraint and a resource sharing constraint. These sets are all children of a set with just an independent placement constraint. There is no need to force simultaneous execution at that level. This is illustrated in Fig. 3.7a. The alternative is to create classes of tasks with roughly identical behavior regarding a resource, and then to execute classes simultaneously that complement each other. Thus, there is a coscheduled set for each class with either a resource non-sharing constraint or an independent placement constraint with an explicit maximal concurrency constraint. These sets are combined by a set with a coscheduling constraint. This case is illustrated in Fig. 3.7b.

Both of these approaches have their advantages and disadvantages. The first variant allows a more fine-grained control, which is useful when a classification scheme is unreasonable for some reason or resource usage can be measured very accurately. The second variant trades this control and accuracy for less synchronization and less overhead at runtime. Of course, one can create arbitrary solutions that lay somewhere between these two. For instance, one could have multiple sets of classes with each set yielding good combinations.

Overall, it is possible to construct a coscheduled set for every use case. Table 3.1 gives an overview, which constraints would normally be utilized for which use case. Due to the modular nature of this coscheduling model, it is easy to have several use cases side by side, such as parallel applications being coscheduled and normal tasks being scheduled resource-aware. The building blocks can simply be combined in a coscheduled set with an independent placement constraint. Similarly, nesting of use cases, such as resource-aware placement of tasks within a coscheduled parallel application or resource-aware placement of multiple coscheduled parallel applications, can be expressed equally simple by nesting the building blocks.

Table 3.1: Use cases and their typically required constraints. If not all instantiations of a use case typically require a constraint, this is indicated by brackets. Not shown is the alternative realization of contention use cases described in the text, that is more similar to the optimization use cases.

Use Case	Coscheduling	Minimal Parallelism	Minimal Concurrency	Maximal Concurrency	Close Placement	Independent Placement	Resource Sharing	Resource Non-Sharing	Resource Isolation
Application Design									
Fine-grained synchronization	✓	(✓)			✓				
Scalability of algorithms	✓	(✓)			✓				
Response time	✓				✓				(✓)
Lock holder preemption	✓	(✓)							
Static load balancing	✓	(✓)							
Auxiliary tasks	✓				(✓)		(✓)		(✓)
Execution unit optimizations	✓						✓		✓
Cache optimizations	✓						✓		✓
Resource Management									
Memory pressure	(✓)		(✓)				✓		✓
Cache pressure	(✓)		(✓)				✓		✓
Execution unit contention				(✓)		(✓)		(✓)	
Cache/memory bandw. contention				(✓)		(✓)		(✓)	
Temperature balancing				(✓)		(✓)		(✓)	
Power capping				(✓)		(✓)		(✓)	
Energy contention							✓		✓
System Design									
Parallel application management	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)
Concurrency management		(✓)	(✓)	(✓)					
Affinity management	(✓)				(✓)		(✓)		
Side-channel elimination									✓
Exclusive device access							✓		✓
Expose system functionality							✓		✓
Special architectures							✓		✓

3.6 Theory vs. Practice

The scheduling constraints attached to coscheduled sets carry certain guarantees on which use cases rely. Unfortunately, it is unlikely – if not downright impossible – for a coscheduler to adhere to each and every constraint by the letter. A coscheduler might support only a subset of constraints, constraints are not realized exactly as described, or constraints might get violated from time to time. In some cases, this is of critical importance for a use case; and sometimes it would just have been nice for the extra percent of performance. In particular, the following constraint violations could happen more or less often:

Fragmented execution: There are less SEs running simultaneously than there should be.

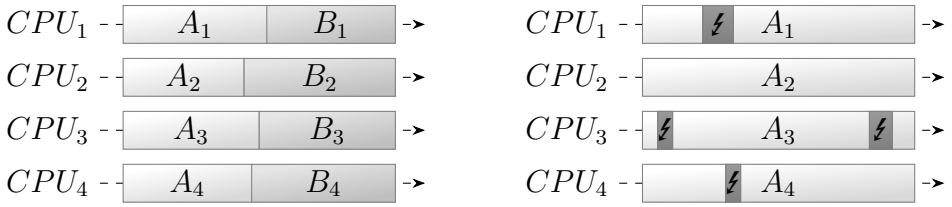
Overstretched execution: There are more SEs running simultaneously than there should be.

Misaligned execution: The SEs are running on a set of CPUs, which does not satisfy a placement constraint.

Non-isolated execution: Other SEs are interfering with running SEs although this was forbidden.

One violation that is almost impossible to avoid completely is fragmented execution. When the coscheduler decides to execute a coscheduled set, its tasks must transition from ready to running. Unless there is a barrier of some kind involved, this transition will happen more or less gradually and might also include a short phase of non-isolated execution as depicted in Fig. 3.8a. Similar reasoning can be applied to blocking tasks within a set with a minimal parallelism constraint, where this information needs to be carried to other CPUs, or handling of an interrupt on a CPU (cf. Fig. 3.8b). In both cases, there are short moments where constraints are violated. While nothing can be done about message latency, the effect of interrupts can be mitigated by disabling or synchronizing their activities. But that is neither always possible nor always necessary. (Note, that fragmented execution may also occur, when there are too many SEs in a set with a coscheduling constraint. But as this usually points to an issue with the use case and not the coscheduler, it is ignored here.)

Without quantifying it exactly, it is possible to distinguish three levels of adherence to a constraint, that can be realized by a coscheduler or demanded by a use case: relaxed, strict, and precise. The default in this thesis is a *strict adherence* to constraints, where only (very) short term violations during context switches, interrupts, or other reconfigurations are allowed. A *precise adherence* forbids even those, while under a *relaxed adherence* constraints might be violated for longer periods of time, i. e., the guarantees are turned into best-effort approaches. The less exact the adherence, the less costly is usually its realization and the more freedom for an implementation exists.



(a) Gradual context switch from A to B or message latency when A transitions to blocked after A_2 blocks.

(b) Interrupt handling.

Figure 3.8: Typical sources of transient fragmented execution.

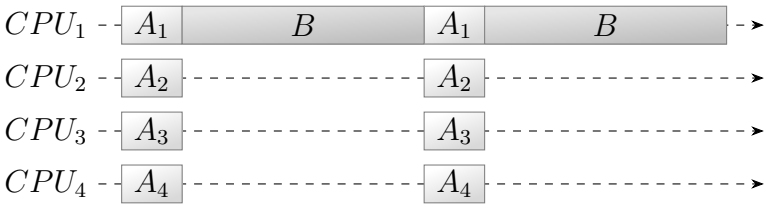


Figure 3.9: Potentially unreasonable adherence to A's coscheduling constraint. Ignoring it, would more than double the usable CPU time. Unless A experiences a severe performance drop or energy consumption is important, A should not be coscheduled.

If a use case does not get its desired guarantees, potential consequences range from often more annoying issues, such as an increased energy consumption, to more noticeable problems – usually performance issues. Performance-wise, the worst case is a standstill as seen with active waiting or livelocks: spending CPU cycles without making any progress. Certain use cases might even fail or produce errors, when a constraint is not fulfilled, such as an operating system panicking when executed within a multicore VM because of not properly synchronized vCPUs. Other potential issues are security concerns: maybe an isolation constraint is used to close certain side-channels and a violation of that constraint might allow an attacker to compromise the system.

In the end, the operating system scheduler has to trade constraint adherences on a case by case basis against the overall scheduling goal. For instance a slightly decreased performance or increased energy consumption of one application might be tolerable, when this decision removes fragmentation, indirectly gaining more profit overall. The example in Fig. 3.9 shows a system with a coscheduled parallel application and an application consisting of a single task. CPU time is distributed fair, each application receives 50%. Violating the coscheduling constraint would more than double the distributable CPU time. If one does not care about the increased energy consumption, this would be a worthwhile trade-off, unless the performance drop of the formerly coscheduled application is severe.

3.7 Chapter Notes

The coscheduling model presented in this chapter is picked up in Chapter 4, where the capabilities of several existing coscheduling approaches are assessed, and in Chapter 6, where an algorithmic framework is laid out, which realizes this model. The question of how the operating system can actually manage an arbitrary agglomeration of coscheduled sets fairly and efficiently (including necessary trade-offs) is considered later as part of Chapter 9. A specific nested use case, namely the resource-aware placement of whole parallel applications, is discussed later as part of Chapter 11.

With respect to application failures caused by an absence of coscheduling, I personally experienced Linux VMs failing to boot some secondary CPUs and FreeBSD VMs panicking when the skew in progress of vCPUs becomes too large in the wrong moment. Mind you, that was during regular usage without actually trying to force anything.

Chapter 4

Coscheduling Approaches

This chapter presents several existing coscheduling approaches. Each approach is analyzed and put into perspective with respect to the definitions given in Chapter 3. Specifically, we look at the supported scheduling constraints and what that means in terms of versatility.

This starts in Sections 4.1 and 4.2 with Ousterhout's coscheduling algorithms and Feitelson's gang scheduling approach, respectively. These two have basically created a research area of its own. Section 4.3 presents another approach that was created specifically with clusters in mind. Then, more different from the approaches so far, the coscheduler within VMware ESXi is examined in Section 4.4. Finally, in Section 4.5 some research on top of these approaches is briefly discussed.

Please note, that some additional scheduling approaches – which do not exactly realize a form of coscheduling – are discussed later as part of Chapters 9 and 10.

4.1 Ousterhout's Coscheduling

In his work about coscheduling [1], Ousterhout defines three algorithms, that try to achieve coscheduling according to his definition. While the design proposed by this thesis differs greatly from Ousterhout's approach, his work can be considered as the start of the research into coscheduling.

The *Matrix method* uses, what is called by other authors today, an *Ousterhout matrix*. In this matrix CPUs are represented as columns and time-slices as rows. While the number of columns is fixed, the number of rows can vary dynamically depending on the load in the system. A group of tasks is placed in the first row that has enough free entries to hold all tasks of that group – no matter whether they are runnable or not. Scheduling is then globally synchronized, executing the rows round-robin, one row at a time. Every CPU executes the task from the current row in its own column. In cases when said task is not runnable or blocks during execution, the CPU cyclically scans its own column for an alternate task to execute.

Applying the terminology from Chapter 3, the Matrix method supports only flat coscheduled sets without any placement constraints. From the time constraints only the coscheduling constraint is supported – and that in a relaxed manner due to the additional "out-of-order execution" of tasks when the original task is not runnable.

The *Continuous algorithm* unravels the matrix row-wise yielding a long array. This array is then always accessed by considering sliding windows with a size equal to the original row length, that is, a window may span two rows within the original matrix representation still with one entry per CPU. A group of tasks is placed within the array by sliding a window, until the window has enough free, not necessarily consecutive elements to hold the whole group. For scheduling tasks, a window is advanced every time-slice so that the leftmost element in the window is the leftmost element of a group of tasks that has not yet been coscheduled with maximum concurrency in the current pass through the array. That window is then executed. For elements with no runnable task, the alternate selection of the matrix method is used, letting a CPU scan through what would be its column in the original representation. The *Undivided algorithm* is similar to the Continuous algorithm with the difference that groups are only placed in consecutive elements within the window.

These two algorithms are identical to the Matrix method with respect to their coscheduling ability, though they are a bit less relaxed. Also, they distribute CPU time more fairly. Depending on the mapping of CPUs to column indices within the matrix, the Undivided algorithm can be used to realize relaxed close and independent placement constraints.

4.2 Distributed Hierarchical Control

Distributed Hierarchical Control (DHC) is an approach described by Feitelson and Rudolph [43] to achieve coscheduling. DHC shares a few similarities with the coscheduler design proposed in this thesis, yet it differs in others. It was proposed in 1990 targeting early parallel machines and was later adapted to work on clusters [69]. An adaptation towards multicore systems with their unique properties and today's software requirements was not done. Due to some similarities, this thesis could be considered as this adaptation.

Similar to the approach proposed in this thesis, DHC also uses a hierarchical structure internally, albeit it is constructed differently. DHC's structure is less flexible and ignores the topology of the system. It mainly considers scalability of the approach itself, without considering the scalability of the executed software. Initially, fairness was barely considered, which was addressed later [70]. Still, the proposed fairness is skewed towards smaller gangs, and a fair distribution of CPU time at application level – which is considered essential in this thesis – is explicitly discarded. DHC does not support any nesting and interactivity is not considered. Also blocking tasks and dynamic group sizes are ignored.

DHC builds a binary tree of controllers on top of the CPUs of the system. Each controller is responsible for the controllers underneath and, thus, for a certain number of CPUs. Each controller is associated with a runqueue, that holds gangs which just fit this controller but not a child controller. Each controller is either active, disabled, or idle. An active controller schedules its gangs actively and can control the state of child controllers. A disabled controller simply does as it is told by the parent, i. e., it schedules a part of a gang somewhere above it in the hierarchy. An idle controller waits for the end of the current scheduling round. A new scheduling round is started by making the root controller active. This controller then disables all child controllers and schedules each of its gangs for a time slice. After that, it makes its children active and becomes idle itself. When a gang does not span all CPUs, corresponding controllers can be activated, so that this otherwise wasted computing power is spent on smaller gangs. This is called *selective disabling* and causes a fairness skew towards smaller gangs.

Noteworthy is, that it is explicitly left open whether the controller network is a logical or an actual physical network. Realizing the controller network separately from the CPUs of the system provides two advantages: there is less interference with user code, and – due to the tree being balanced – scheduling decision arrive at all affected CPUs at the same time, independent of the network latency. The disadvantage is its static structure once created.

DHC considers basically two types of fairness: giving each application the same amount of system time (called uniform), and giving each task the same amount of system time (called weight). Giving each application the same amount of CPU time is discarded as being too wasteful despite of their selecting disabling, which counters that. Load imbalances between child controllers in the uniform case are solved by favoring gangs in the lesser loaded child compared to all other gangs, i. e., another fairness skew towards smaller gangs. Finally, the weight case favors gangs that fill or almost fill their controller over other gangs.

DHC only balances newly created gangs. Migration is not considered. In the first description [43] of DHC, this balancing or mapping is done decentrally by searching the vicinity of the current controller for the least loaded controller. To support that more efficiently, the binary tree is augmented with additional links, connecting controllers within the same level. This leads, e. g., to an X-tree [71]. To improve selective disabling, individual tasks within a gang are mapped to CPUs so that as many controllers as possible are kept free. The definition of load includes only the load below a controller and ignores imbalances within the controller hierarchy completely. Thus, this method might make decisions that increase the skew of load.

In [70] an alternative is suggested that starts at the root controller and always selects the least loaded child controller. This solves one of the aforementioned problems, but could still increase fragmentation instead of reducing it, when a gang is mapped to a controller whose sibling has no load. A central strategy for mapping applications in DHC is suggested in [72]. Especially, it pays attention to reduce imbalances first, if possible.

In terms of Chapter 3, DHC supports only flat coscheduled sets with a strict coscheduling constraint. Other time constraints are not supported. Though not in the scope originally, limited support for placement constraints can be realized by matching the system topology with the controller hierarchy – possibly generalizing the idea to non-binary trees. Then, depending on the mapping, either the close or independent placement constraint can be realized. Additionally, without selective disabling and an exact match of system topology and controller network, the isolation constraint can be realized.

4.3 Distributed Queue Tree

The Distributed Queue Tree (DQT) approach by Hori et al. [44,45] shares many similarities with DHC. Both are examples for a scheduling class, which Hori et al. dubbed Time Space Sharing Scheduling (TSSS).

At its core, TSSS refers to all those schedulers, which combine time sharing and space sharing, so that partitions of a parallel system are repeatedly allocated to parallel applications. Glossing over a few details, such as global time slots or some requirements for the used communication network, some instances of the scheduling approach in this thesis could be considered to fall into the TSSS category. While TSSS explicitly honors topology, it only supports flat coscheduled sets: multiple partitions are independent from each other. As partitions are assigned exclusively, this realizes the resource sharing and isolation constraints. TSSS targets Feitelson's gang scheduling, that is, a realization of TSSS has to support at least the coscheduling constraint.

DQT as an instantiation of TSSS is effectively DHC with a controller network organized along the system topology, without selective disabling, and with its own set of rules for scheduling and (static) load balancing. The size of a coscheduled set must be known a priori and is then fixed. DQT as a concept makes no further specification on how a parallel application is scheduled within its partition. With the approach targeting uniform multiple instruction/multiple data (MIMD) systems, migration is considered out of scope.

4.4 Coscheduling in VMware ESXi

The hypervisor in VMware's vSphere, named ESX or ESXi depending on the version, is the only realization of a coscheduler in a commercially successful product. The scheduler itself [9,68] has changed drastically across versions. With its various configuration possibilities a description capturing all of its functionality is out of the scope, here. Instead, we will focus on just a few specific aspects of the coscheduler.

Since ESX 2.x a relaxed coscheduling constraint is supported. It started out quite strict – but not fully meeting the requirement for strict constraints given in Chapter 3 – and got more and more relaxed since then. Initially in ESX 2.x,

runnable vCPUs of a VM had to start execution synchronously but were allowed to get preempted independently – unless the preempted vCPU was lagging too far behind, in which case all vCPUs were preempted. With ESX 3.x this was relaxed and – while coscheduling is still maximized when possible and sets with more tasks than CPUs are still forbidden – the constraint is now more akin to a minimal concurrency constraint: at least the vCPUs, which are lagging behind, have to be coscheduled. Scheduling more vCPUs is just considered a bonus. In ESX 4.x this was converted again: instead of forcing the execution of dawdling vCPUs, fast-paced vCPUs are now prevented from getting executed. That is, instead of a minimal concurrency constraint, a selective maximal concurrency constraint is applied until the other vCPUs have caught up. Note, that together with these changes, there were also changes with respect to what is considered progress for a vCPU, but these details do not matter for the high-level overview here.

ESXi has also support for placement constraints, though the supported constraints also vary across versions. Up to ESX 3.x, there was the concept of *scheduler cells*, which are identically sized groups of (physical) CPUs, which (usually) honor the system topology. Thus, they represent a (real or artificially inserted) level in the topology tree. ESX 3.x does not allow mapping of coscheduled sets to topological units above the scheduler cells, hence, coscheduled sets implicitly have strict resource sharing constraints. ESX 4.0 abolished scheduler cells and adopted a relaxed independent placement policy of coscheduled sets within topological units below NUMA domains and (since ESX 4.1) a strict close placement policy for coscheduled sets above NUMA domains. Optionally, the former can be replaced by a relaxed close placement constraint. With ESXi 5.x the load balancing algorithm was overhauled. While the high-level placement goals of coscheduled sets remain the same, the placement has become slightly more contention-aware. Also, it is now possible to expose the NUMA topology of a coscheduled set to the VM it represents – an example for nested coscheduled sets.

4.5 Further Considerations

All presented approaches require an explicit notion of coscheduled sets. In their originally intended application area these are easily identified: one coscheduled set per parallel application. However, with all use cases and today's general purpose systems in mind, this is no longer good enough. Basically, research branches out in three different directions from here.

Firstly, it is possible to generate coscheduled sets automatically. This can range from the more simplistic auto-grouping done by Linux for accounting purposes over automatic grouping [73] or topology-aware nesting [74] via instrumented parallel programming environments to the wide area of resource-aware scheduling (cf. Section 2.2). With respect to this thesis, the automatic gener-

ation of coscheduled sets is considered out-of-scope – though the integration of such approaches is discussed later in Chapter 11.

Secondly, with given coscheduled sets it is still possible to drive their time and placement constraints dynamically at runtime. This is of particular interest, when coscheduled sets were the result of some more primitive heuristic or when a set changes its behavior and it is not desired to reform the sets in the system. For example, in *flexible coscheduling* [75, 76] the coscheduling constraint is tied to a feedback loop monitoring the set. Similarly, this can also be done for placement constraints, as the ADAPT framework [77] demonstrates.

Thirdly, there is research which addresses shortcomings of the presented approaches. For instance, there is *paired gang scheduling* [78], which addresses missing support for interactive workloads by pairwise merging of I/O intensive and CPU intensive coscheduled sets – and losing the ability for strict coscheduling along the way. With *controlled contention* [79] the absence of dynamic placement constraints is compensated by opportunistically merging coscheduled sets until enough load has been accumulated to keep a partition busy. Another example is *balanced scheduling* [80], a probabilistic (i. e., very relaxed) coscheduling approach, which avoids synchronization by ditching time constraints completely and relying only on placement constraints, so that no two tasks within a set end up on the same CPU. Strictly speaking, this thesis itself also falls into the last category: unify previously independent research and provide an approach, which is ready for today's workloads on contemporary systems.

4.6 Chapter Notes

This concludes the introductory part of this thesis. The second part presents my coscheduling approach and its realization, which addresses shortcomings of the approaches presented here and also unifies other scheduling ideas. Later, as part of individual chapters of the third part, more specialized scheduling approaches will be discussed and their relation to my approach.

Part II

Coscheduler Design

where a new coscheduler is motivated, designed, implemented, and evaluated

Table of Contents

Chapter 5

Design Rationales

Looking at existing approaches of creating a scheduler capable of coscheduling, they can be grouped into a few different categories. Many early or ad-hoc coscheduling approaches are quite inflexible: they can only coschedule across the whole system, are not able to schedule tasks simultaneously that belong to different applications, or lack support to handle legacy situations. Approaches developed for specific use cases, such as the virtual machines or the avoidance of resource contention, have a quite limited scope, which often makes them unsuitable for other use cases. Some approaches require support by applications themselves, which makes them unsuitable for unmodified applications. While drawbacks can be addressed individually, these modified solutions often also introduce different restrictions, which can be limiting in other scenarios.

A new coscheduler shall be devoid of these drawbacks, motivating the following design rationales.

5.1 Versatility

The coscheduler shall support a wide variety of use cases. There are just too many possibilities to apply coscheduling. Supporting only a subset of use cases will lead to a situation, where only parts of a system or only a few applications are able to gain benefits. Other coscheduling-based optimizations are prevented and applications are barred from realizing more efficient solutions. Overall, a system will not be able to reach an optimal operating point with a limited coscheduler. This also means that the coscheduler has to support different use cases simultaneously. The realization of one use case should not prevent the realization of different use cases in a different part of the system, at a different time, or a different abstraction level.

Versatility is achieved, when arbitrary, possibly nested coscheduled sets can be specified and every scheduling constraint is supported at least in its strict or even precise variant. Ideally, adherence levels can be specified at constraint level, giving the scheduler more optimization opportunities.

5.2 Scalability

With "multicore" slowly transitioning to "manycore", the coscheduler shall be scalable. Scalability is a delicate topic for coscheduling: experience with clusters has shown, that the collective context switch can become problematic when scaling up – latency gets higher and simultaneousness suffers. For clusters, this was addressed in various way: the development of implicit scheduling techniques, relying on synchronized clocks, or the introduction of hardware support. Each solution has its own drawbacks: the first limits versatility, the second cannot handle spontaneous events, and the third is not available in every cluster. Luckily, communication within multicore systems is much less expensive than in clusters. That is, while technically not true, multicore systems can be considered to have integrated support for synchronized context switches in the form of inter processor interrupts (IPIs).

For coscheduling on larger systems, there are two types of scalability to consider. On the one hand, performance of identically sized partitions should not suffer, just because a system is larger. On the other hand, it should be possible to create larger coscheduled sets on larger systems without running into some upper limit.

Scalability is achieved, when coscheduled sets are unrestricted and operation costs are independent from the size of the system. Operation costs may, however, depend on the size of coscheduled sets. Though, even the largest coscheduled set should still be handled without prohibitive overhead.

5.3 Interactivity

Interactive behavior is considered by many coscheduling approaches as a second class citizen: cross application synchronization of time-slices, rigid schedules (e. g., Ousterhout matrix), and from a today's viewpoint ridiculous large time-slices (sometimes measured in minutes) do not work well together with interactivity requirements. This is unfortunate as many multicore systems today are used interactively: from desktops to servers to mobile devices.

The coscheduler shall impose no restrictions on interactive usage. Interactive (or I/O intensive) tasks should still be able to be executed on a rather short notice, as it is accomplished by most scheduling strategies, so that short response times are realized. Note, that interactive behavior is not limited to sequential tasks: whole coscheduled sets might wake up for short parallel execution phases.

Interactivity is achieved, when interactive scenarios can be realized without crippling versatility.

5.4 Non-intrusiveness

Finally, the last rationale shifts the point of view from users or administrators to the developers of schedulers.

The coscheduler shall integrate itself nicely into existing schedulers (or allow a maximum degree of freedom for yet-to-be-written schedulers). That is, in order to add the feature of coscheduling to an existing scheduler, it should not be necessary to entirely replace or rewrite the existing scheduler. Ideally, all features of said scheduler stay intact and the only change is its sudden coscheduling capability. For new schedulers, there shall be no imposed restrictions regarding, e. g., the choice of runqueue or balancing algorithms.

Non-intrusiveness is achieved, when all normally existing scheduling features can still be realized (and possibly existing code reused). This especially means, that legacy use cases, which might have been tailored towards the original scheduler, are still fully supported – just as before. A non-intrusive coscheduler is appealing for users and developers alike, because it keeps proven code intact and allows to introduce the new behavior gradually.

5.5 Chapter Notes

The rationales presented in this chapter form the guidelines for my coscheduler that is designed, implemented, and analyzed in the following three chapters.

The rationales are all motivated by my own experiences when I got exposed to the topic as part of my research on the influence of scheduling on energy consumption on contemporary processors. At some point coscheduling came up as a mechanism, that – theoretically – should be beneficial in the considered scenario, if used just right. But how do you actually prove it practically? No currently available operating system supports coscheduling, so the simple route does not work. Then you look at other people's work: for this one, you would have to express your workload in form of virtual machines introducing another bunch of problems; that one looks promising but cannot handle quickly changing requirements; the other one is able to do that, but cannot handle parallel applications, and so on. Many good ideas that – while solving the issue they were designed for – could not be applied to my problem. So, what's next? Add my own specialized solution to the pool and move on? Or spent some more time and actually do something about it?

I decided to do something about it and shifted my research focus on coscheduling itself. The result so far is captured in this thesis. The original problem is now a small part of it and presented later in Chapter 10.

Chapter 6

Topology-aware Coscheduling

Based on the previously described design rationales, this chapter presents a new design for a coscheduler. While the overall goal is the same as of any coscheduler, namely executing certain tasks simultaneously, the design presented here differs from previously presented designs substantially. The major reason for this difference is that it does not try to solve a single problem with coscheduling, but that it is able to handle most – if not all – use cases one can think of on current multicore architectures.

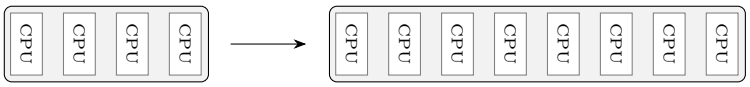
This chapter starts with a description of the general structure of the design and the employed algorithms in Section 6.1. After this, the construction of coscheduled sets within that structure is discussed in Section 6.2. Section 6.3 covers the load balancing of these coscheduled sets. After that, different possibilities to handle fragmentation are subject of Section 6.4. The chapter closes with an alternative accounting realization, which eliminates the remaining fairness and balancing issues in Section 6.5.

6.1 Basic Structure

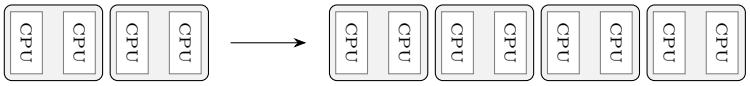
To facilitate a scalable coscheduler, the core of the design consists of the concept of *synchonization domains (SDs)* and rules on how they are created, linked, and selected. In a sense, synchronization domains are partitions on steroids. Synchronization only happens within a SD. Thus, independent SDs operate autonomously in a large system.

6.1.1 Synchronization Domains

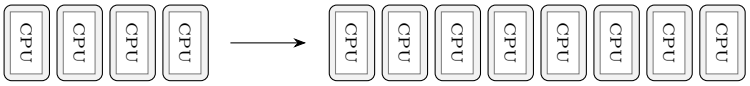
A SD is described by a set of CPUs. In that sense, it is similar to a partition. However, it is also more than that. A SD is scheduled and in turn also schedules other SDs in its place. So, the structure of SDs goes beyond partitioning schemes as described in, e. g., [65]. Synchronization activities by the scheduler



(a) One SD at system level (e. g., Ousterhout matrix). Arbitrarily large coscheduled sets, but synchronization does not scale.



(b) One SD per processor (e. g., VMware ESX 3.x). Sacrifice maximal set size for bounded synchronization costs.



(c) One SD per CPU (e. g., Linux Completely Fair Scheduler). No coscheduling, no synchronization.

Figure 6.1: Scaling a fictional system with dual-core processors using differently sized SDs. There is a trade-off between synchronization costs and the maximal size of a coscheduled set. To get the best of both worlds, SDs of different sizes must be supported at the same time with transparent switching between them (not shown).

are confined to the boundaries of a SD. On the one hand, this means that independent SDs make progress on their own. On the other hand, it means that coscheduling can only happen within a SD because of the required simultaneous context switch.

Setups supporting only a single, system-wide synchronization domain (e. g., everything based on an Ousterhout matrix) are inherently unscalable, as synchronization overhead within the SD will increase with larger systems. The other extreme is support for small SDs only, such as one SD per processor. While this does not cause increased synchronization overheads on larger systems, it prevents the creation of larger coscheduled sets. Operating systems without support for coscheduling can be seen as supporting SDs with just one CPU per SD. These three cases are illustrated for a fictional system with dual-core processors in Fig. 6.1. Thus, a versatile system must be able to support large and small SDs, and it must be able to switch between them during runtime. And if interactivity is a criterion, this switching must happen quite fast and on-demand.

Each SD comes with its own runqueue and does its own scheduling. Theoretically, different synchronization domains could even utilize different scheduling algorithms. A scheduling decision within a SD affects exactly those CPUs, which are part of the SD; not more, not less. Elements within a runqueue are usually references to other, probably smaller SDs – except for element within SDs spanning only a single CPU, which may also reference tasks. Coscheduled sets are

mapped to synchronization domains. When there are no suitable SDs, new SDs must be created and integrated into the already existing SD structure.

Initially, the system contains one system-wide SD, which is empty, i. e., the system is idle. Further SDs are added as children of existing SDs. The CPUs of a child SD must be a subset of the CPUs of the parent SD. If a child SD does not cover all CPUs of the parent, it must be part of group of multiple child SDs that are pairwise disjoint and cover all CPUs of the parent. This leads to a tree-like structure of progressively smaller synchronization domains. Within this SD structure, it is possible to differentiate two types of parent/child relationships:

Vertical relationships: A vertical relationship splits a larger SD into multiple smaller SDs. SDs in a vertical relationship always belong to the same coscheduled set. The tree-like representation of a system's topology gives a good idea of vertical relationships.

Horizontal relationships: A horizontal relationship connects SDs belonging to different coscheduled sets. Without loss of generality, we assume in this thesis, that SDs in a horizontal relationship are of the same size, which simplifies the upcoming descriptions and diagrams.

With this hierarchy of progressively smaller SDs, synchronization within the system is reduced to the necessary minimum. The deeper SDs are within the hierarchy, the less CPUs have to be synchronized during context switches within those SDs.

6.1.2 Scheduling

Each SD has its own runqueue. This runqueue is used to coordinate child SDs and – when the SD is responsible for only one CPU – tasks. Each element in the runqueue references work for all CPUs of the SD. That is, child SDs of a smaller size are only entered in form of a group. To determine which task should be executed on a certain CPU, the SD hierarchy is processed from top to bottom. In each SD, one element from within the runqueue is picked and the selection process continues recursively with the SDs represented by this element. This continues until leaf SDs are reached, where finally tasks are selected. Every picked SD along the paths back to the top is considered running.

Contention is avoided by designating a CPU as *master* for each SD, which is responsible for coordinating the corresponding CPUs. Being master is not a fixed role; instead, this role can be freely transferred between CPUs belonging to the SD. The master examines the runqueue, picks children to execute, and notifies masters of selected children, so that they will do their part of the task selection. The master is also responsible to enforce preemption after the time-slice has been used up. Every CPU is the master of some SDs – a least of those covering just the CPU in question.

Each CPU holds a reference to the top-most running SD for which it currently has master responsibilities; this SD is the CPU's *personal root* within the SD

hierarchy. When a CPU receives the proper notification (e. g., via IPI for a preemption, or via timer interrupt for a used-up time-slice), it starts a task selection at its personal root: the CPU picks an element from the runqueue, updates the personal roots of master CPUs (other than itself) of child SDs referenced by the picked element, and notifies those masters afterwards, so that they start a task selection of their own, in addition to continuing downwards itself. This way, the selection process is decentralized and triggered on-demand.

When there is only one element in the upper levels of the SD hierarchy and when there are no changes, there is no need to examine them repeatedly. Hence, they do not cause overhead, when they are not actively required. If at any point in the hierarchy an empty runqueue is found, it means that the CPUs of the SD in question are idle. If load balancing is also unable to find some work and no other techniques are applied (see Section 6.4), the CPUs of the SD must be notified so that they preempt the currently running task in favor of an idle loop and possibly entering a power save mode.

While selecting a task proceeds from top to bottom, enqueuing and dequeuing of tasks proceeds from bottom to top of the SD hierarchy. When a task becomes runnable, it is inserted into a leaf SD. This in turn might make the SD itself runnable. When that is the case, the group containing this SD is inserted into the parent. This process is repeated, until the top is reached. As higher levels in the SD hierarchy represent multiple CPUs, it is likely that this process encounters a SD that is already inserted into its parent runqueue. In this case, the process is stopped early. Thus, enqueuing does usually not traverse the whole hierarchy. Dequeuing works similarly. The task is first removed from its leaf SD. If the leaf SD itself can no longer be considered runnable, it is removed from its parent – and so on.

The process of enqueuing and dequeuing a task might trigger a preemption. Either because a SD becomes runnable that is more important than the currently running one, because the currently running tasks belong to a no longer runnable SD, or because of a change in importance due to the enqueuing or dequeuing. Performing these preemption checks might require to traverse the hierarchy further, depending on the actually employed preemption rules. If a CPU considers a preemption to be advisable, this is a good point for the CPU to take over the master role and trigger the preemption itself rather than notifying the current master first.

6.1.3 Fairness

Achieving fairness and load balancing in this kind of setup is less trivial than in other scheduling schemes. There are several issues to address, foremost concerning fairness. When the fairness question is answered, the load balancing is more or less a result of this.

Because of the possibility of a scheduling entity representing varying amounts of sequential or parallel applications possibly spanning multiple CPUs, it must

be possible to assign different amounts of CPU time to different SEs. This can be done by assigning a weight to every SE and implement a proportional-share scheduling. By controlling this weight, a single entity can represent various amounts of load and it is also possible to realize different interpretations of fairness. Thus, the proposed scheme is similar in spirit to Waldspurger's stride scheduling [81, 82] or the Linux Completely Fair Scheduler [83]. Though the similarities end at the hierarchical multicore setup. This thesis uses the following naming conventions:

Definition 14 (Weight of a scheduling entity)**.** Each scheduling entity se has an associated weight w. (The weight itself depends on the type of SE and will be concretized later.)

$$weight(se) = w$$

Definition 15 (Weight of a runqueue)**.** The weight of a runqueue rq is the sum of weights of all enqueued scheduling entities se_i.

$$weight(rq) = \sum_{se_i \in rq} weight(se_i)$$

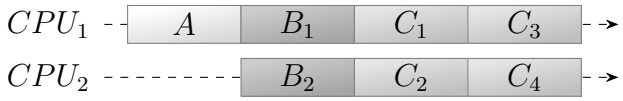
Definition 16 (Share of a scheduling entity)**.** The share of a scheduling entity se_i is its fraction of the total weight of the runqueue rq, where it is enqueued in.

$$share(se_i) = \frac{weight(se_i)}{weight(rq)}$$

The sum of all shares of a particular runqueue is one. The time, that a runqueue receives (because its SD was picked), is distributed among the enqueued entities according to this share. The method, how the time distribution is actually realized, does not really matter. For instance, Waldspurger modifies the task selection frequency, while Linux additionally uses differently sized time-slices.

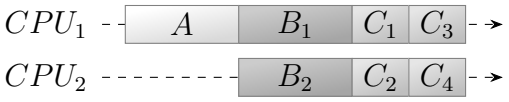
Every runqueue (except the top runqueue of the root set) is represented by another scheduling entity in its parent SD. For horizontal connections, this is a one-to-one mapping, for vertical connections a many-to-one mapping, i. e., a scheduling entity represents multiple runqueues. The weight of such a SE depends on what it represents and what is considered fair. A SE may represent: a single task, a parallel application (consisting of multiple SEs), or a bunch of unrelated sequential or parallel applications. In case it represents multiple SEs, these might or might not be distributed over multiple CPUs. As an extension, a SE might also represent only a fraction of a parallel application. These SE types are confronted with different types of fairness. Typically it is either fairness at task level, fairness at application level with respect to CPU time, or fairness at application level with respect to system time. (Other kinds of fairness, such as fairness at user level, can be modeled with this, too: the *user* is then simply an artificial application consisting of multiple other SEs.)

The results of applying all three types for fairness in different situations are depicted in Fig. 6.2. Fairness at task level (or SE level) is appropriate, when the



(a) Fairness at task level: every task gets the same amount of CPU time.

(b) Fairness at group level with respect to CPU time: every group gets the same amount of CPU time.



(c) Fairness at group level with respect to system time: every group gets the same amount of system time.

Figure 6.2: Different types of fairness applied to certain groups of SEs. Group A: a single task. Group B: multiple tasks, but not more tasks than CPUs. Group C: multiple tasks, more tasks than CPUs.

SEs in a group are independent from each other. Within the SD structure, this is the case for vertical connections, where a SE represents multiple independent runqueues, each with possibly multiple independent SEs. Fairness at group level is appropriate, when there is a close relationship between the SEs within the group, e. g., parallel applications. A fair distribution of CPU time among groups is useful for malleable applications: the computational resources each application receives are the same, no matter how many tasks. With typically sub-linear speedups, this promotes the idea of using as few CPUs as possible, unless there are some circumstances which allow a higher performance, such as a higher amount of combined cache. This is exploited later in Chapter 9. A fair distribution of system time among groups can be considered traditional, as this is what all partitioning approaches do as well as Ousterhout's matrix method and Feitelson's Distributed Hierarchical Control (cf. Chapter 4). It provides size-independent fairness for rigid applications without having to resort to weight manipulations. In the context of this thesis, it has no use.

Task level fairness is achieved by an *aggregating SE*. An aggregating SE is transparent from a CPU time perspective: a share of any of the represented SEs is directly comparable to a share of a SE enqueued in the same runqueue as the aggregating SE. Equal shares at both levels indicate that both SEs receive an equal amount of CPU time (given perfectly balanced load). To achieve this, the weight of the aggregating SE must be equal to the sum of all represented weights.

Definition 17 (Weight of an aggregating SE)**.** The weight of an aggregating SE *se* is the sum of weights of all runqueues rq_i represented by this SE.

$$weight(se) = \sum_{rq_i \in se} weight(rq_i)$$

Note, that this is only exact, when there is no load imbalance. When an aggregating SE is picked during scheduling, every runqueue represented by it gets the same amount of CPU time. Hence, if the load is not distributed evenly (e. g., one CPU with three tasks and one with two tasks), some SEs will receive more CPU time than indicated by their share and others less. Still, in this case it is sufficient to address the imbalance only within the group of represented runqueues by, e. g., periodic rebalancing. This is different, when there is not enough load to occupy all CPUs. Specifically, the situation shown in Fig. 6.2a would have been realized differently: assuming a weight of one for each task, the weight of an aggregating SE for group *A* is 1, for *B* 2, and for *C* 4. Thus, *A* would get only half of the depicted time with the above formula, and no rebalancing within the group would be able to address that. That is, fairness is skewed towards larger applications. One possibility to resolve this, is to account for empty runqueues as done in the following revised definition.

Definition 18 (Weight of an aggregating SE, revised)**.** The weight of an aggregating SE *se* is the sum of weights of all non-empty runqueues rq_i represented by this SE scaled to the overall number of runqueues.

$$weight(se) = \frac{|\{rq \in se\}|}{|\{rq \in se : rq \text{ non-empty}\}|} \cdot \sum_{rq_i \in se} weight(rq_i)$$

Another possibility to handle this problem is to incorporate the ability to handle any differences of expected and received CPU time into the scheduling algorithm itself, which is discussed later in Section 6.5. It should be noted, that any way to ensure fairness in such a case also increases fragmentation. Therefore, the scheduler should make sure to not get into such a situation in the first place (cf. related discussion in Chapter 9).

Fairness at application level is achieved by *group SEs*, which are used for horizontal connections. In the simplest case, an application is represented by just one SE. Then, the weight just depends on the desired type of fairness.

Definition 19 (Weight of a group SE (fair CPU time))**.** The weight of a group SE *se* for a fair distribution of CPU time is a constant weight w_c, completely ignoring the SEs it represents and where in the SD hierarchy it is enqueued.

$$weight(se) = w_c$$

Group SEs with identical weights receive the same amount of CPU time, given a perfectly balanced load. If a fair distribution of system time is desired instead, the weight has to be scaled with the number of CPUs, i. e., the "system" the group SE is associated with.

Definition 20 (Weight of a group SE (fair system time))**.** The weight of a group SE *se* for a fair distribution of system time is a constant weight w_c, scaled to the number of CPUs the SE represents – its capacity.

$$weight(se) = w_c \cdot capacity(se)$$

Until now, a group SE represents something like a parallel application in its entirety. This singular representation is what realizes coscheduling in the end. However, when coscheduling is not necessary, or only smaller sub-groups needs to be coscheduled, an application can also be represented by multiple group SEs. This complicates the weight calculation to retain fairness at application level, because load represented by different group SEs of one application does not need to be balanced; only within each represented group a balance of load is necessary. This act of breaking up a group SE creates multiple *split SEs*, which are enqueued in its stead. The weight of a split SE depends – among other things – on the load it represents. Load itself is simply the aggregation of all weights as for the weight of an aggregating SE.

Definition 21 (Load of a SE)**.** The load of a SE *se* is the sum of weights of all runqueues rq_i represented by this SE.

$$load(se) = \sum_{rq_i \in se} weight(rq_i)$$

Definition 22 (Weight of split SEs)**.** A split SE se_i represents a subset of some other (imaginary) SE *se*. Its weight is the corresponding fraction of the weight of the SE *se*.

$$weight(se_i) = weight(se) \cdot \frac{load(se_i)}{load(se)}$$

Note, that the sum of weights of all split SEs se_i is equal to the weight of the original SE *se*.

$$\sum_{se_i} weight(se_i) = weight(se)$$

The result of splitting a group SE is – except for the hierarchical aspect – similar to task groups in Linux. Though, the technique can also be used to enqueue a single SE multiple times in different SD hierarchies by using a predefined fraction of the weight for each split SE. This is needed by one of the fragmentation avoidance techniques described in Section 6.4.

6.2 Mapping of Coscheduled Sets

For each coscheduled set a separate set of properly interconnected SDs is created. This sub-hierarchy is then linked via one or more horizontal connections to the parent set in the existing SD hierarchy. The number and allowed positions

of these horizontal links depend on the scheduling constraints that come with the coscheduled set in question. Time constraints usually limit the number of horizontal connections or they limit horizontal connections to SDs within a certain size range. Placement constraints, on the other hand, usually dictate allowed shapes and positions of SDs.

Within these constraints, the operating system is free to choose the internal layout of the SD hierarchy, so that it can achieve an overall balance and an optimization towards operation goals more easily. In the following, the restrictions on the SD hierarchy induced by scheduling constraints are discussed.

6.2.1 Time constraints

With the exception of the minimal parallelism constraint, which has a somewhat special position, the time constraints on a coscheduled set control the degree of concurrency. Within the SD hierarchy, every horizontal connection from one coscheduled set to its parent represents an isle, where tasks will be executed simultaneously.

Coscheduling Constraint. For a coscheduled set with a coscheduling constraint this means, that there may be at most one horizontal connection to its parent. To make sure that indeed all runnable tasks (or nested sets) are coscheduled, the SD with the horizontal connection must be wide enough to encompass all runnable tasks. To reduce fragmentation, this SD should also be as small as possible, so that there are more scheduling opportunities within the parent set. This means, that the horizontal link is relocated automatically to a more suitable SD, when the number of runnable tasks changes and the currently linked SD is not the best available match anymore.

A relaxed version of this constraint may choose to do this relocation delayed, especially towards a larger SD. This reduces internal fragmentation at the cost of sometimes coscheduling only a subset of tasks within the set.

Maximal Concurrency Constraint. An upper limit on the degree of concurrency is enforced by restricting a coscheduled to a single SD with a width not exceeding the allowed maximum. When there are more tasks than CPUs, time-sharing will be used within leaf SDs to schedule all tasks. Note, that this constraint alone still allows to form horizontal links at will as long as they are below or at the mentioned SD. Having more than one horizontal link cannot guarantee coscheduling of all tasks anymore, but that is not the goal. Instead, using for example leaf SDs within the set keeps synchronization requirements – and with it fragmentation – to a minimum.

Unless an implementation supports arbitrarily sized SDs, not all maximum values can be supported. A relaxed version of this constraint could round the limit upwards to the next supported value, instead of downwards which is necessary for the strict version.

Minimal Concurrency Constraint. A lower limit on the degree of concurrency is achieved by not allowing horizontal links to form between SDs with a width below the minimum. Additionally, horizontal links beyond the first may only be established, when enough tasks (or nested sets) are available to fully occupy all linked SDs. This way, even when only one subset is executed, the constraint is fulfilled. If there is only one link and not enough runnable tasks to even reach the lower limit, it makes sense to handle this constraint like a coscheduling constraint, so that SDs below the limit get used instead of half-empty larger SDs.

Minimal Parallelism Constraint. A minimum degree of parallelism could be realized by maintaining a counter of runnable tasks (or nested sets) within a set. Only when enough tasks are runnable, the set itself is considered runnable and enqueued in the parent set. However, as this constraint is mostly used together with the coscheduling or minimal concurrency constraint, an alternative realization is to declare a set not runnable, once a horizontal connection below the minimum would be established. By simply not establishing this link, the set is dequeued.

6.2.2 Placement constraints

With one exception, placement constraints do not enforce any limit on the number of horizontal links. Instead, placement constraints define allowed layouts of the sub-hierarchy of SDs making up a coscheduled set. So far, given examples always oriented themselves at the system topology. But this is just one way of organizing SDs, which is sufficient for only three of the five defined placements constraints.

Resource Sharing Constraint. The resource sharing constraint is such a constraint, which takes the system topology into account. Specifically, a coscheduled set may only be mapped within a SD, which has all of its CPUs within a matching topological unit (TU). This results in an implicit upper limit of concurrency, but does not prevent arbitrary ways of structuring the SD hierarchy within said TU.

Isolation Constraint. The isolation constraint is similar to the resource sharing constraint. Though, in order to realize the aspect of isolation, it has some additional requirements: Firstly, the target SD must fully encompass a to-be-isolated TU, so that no CPU can be part of a different SD. Secondly and for the same reason, the horizontal link to the parent set must be at this SD, so that no CPUs of the SD can be used for something else. Finally, the scheduler may not perform alternate selections within this SD as discussed later in Section 6.4.

Close Placement Constraint. The close placement constraint is more or less a dynamic application of the resource sharing constraint. The type of TU to be used as an anchor for a coscheduled set with this constraint is determined by the amount of runnable SEs, so that all SEs end up on different sets of CPUs. Generally, the TU in question is just large enough to accommodate all SEs. However, a relaxed implementation might choose to switch to a larger TU earlier than necessary, e. g., because it uses some CPUs of a TU for a different SD.

If sibling TUs are interconnected with a network (e. g., NUMA domains) and only a few are needed, they should be placed near to each other, so that communication costs within the set are minimal.

Resource Non-sharing Constraint. The resource non-sharing constraint cannot make use of SDs aligned to the system topology. Instead, this constraint need the exact opposite: SDs crosswise to the system topology, for example one CPU per processor. Specifically, a SD may have at most one CPU per matching TU. This results in an implicit upper limit of concurrency, but does not have further impacts, similar to the resource sharing constraint.

Independent Placement Constraint. This is a dynamic application of the resource non-sharing constraint, in the same manner that the close placement constraint is the dynamic version of the resource sharing constraint. Thus, the SD selection also depends on the number of runnable SEs. A hierarchy of valid SDs would be aligned to an "inverse system topology". Taking a typical Intel system with SMT and NUMA as example, this inverse topology would start at the system level, branch out according to the number of SMT siblings per core, then branch out according to the number or cores per processor, and finally branch out according to the number of processors or NUMA nodes within the system. As with the close placement constraint, a unit within this inverse topology is selected so that the resulting number of CPUs is just large enough to accommodate all SEs.

Note, that as soon as a coscheduled set with a constraint like this and another set with, e. g., a close placement constraint are nested, the resulting SDs will be aligned neither to the regular nor the inverse system topology. It will be something in between, but following the same ideas.

6.3 Load Balancing

Load balancing must consider the SD hierarchy, as it determines from which runqueues SEs are selected simultaneously and between which runqueues SEs can be moved without upsetting any coscheduled sets and their constraints. This means, that whenever load is moved, it has to stay within the coscheduled set

it came from. However, depending on the actual constraints, there are various options to move that load around.

Looking at a coscheduled set, i. e., a set of SDs with vertical links, the load within that set comes in the form of SEs enqueued in the various SDs. With the exception of SEs used to realize vertical links, the SEs either represent nested coscheduled sets or tasks. Moving load means to dequeue one or more SEs in one SD and enqueue them in other suitable SDs. However, with nested sets and multiple horizontal links, dequeuing a SE and enqueuing it elsewhere in a set, looks different in the parent set. Also, there is the possibility to reestablish a horizontal link between two coscheduled sets at SDs of a different size than before – essentially changing the number of coscheduled CPUs in the set – which has its own peculiar effect on the overall load situation.

Ideally, a state would be reached, where tasks receive their fair amount of CPU time without having to shuffle them around anymore. Phrased differently: if tasks were kept stationary and each task would only receive its fair amount of CPU time, then there would be no undue idle time (a. k. a. fragmentation) in the system. Within the SD hierarchy this global state of being balanced can be broken down: each SD defines an isle where its CPUs execute SEs for the same amount of time. Thus, if each SD is balanced in itself, the whole system is balanced. This allows the realization of a distributed load balancer.

A single SD is balanced, when within each group of vertically linked child SDs the average weight per CPU is the same for every vertically linked child SD. That is, within a vertically linked group, SDs of identical size have the same weight, while a SD with twice as much CPUs as another SD is twice as heavy. More formally, each runqueue (or SE) has a *capacity*, which is – in this thesis – equivalent to the number of CPUs represented by that runqueue (or SE). (Varying the definition of the capacity would allow handling of asymmetric parallel systems.) Within a group of runqueues that is represented by a single SE, the load must be distributed proportionally to that capacity. This leads to the following definition of the targeted load for a runqueue.

Definition 23 (Target load for a runqueue)**.** The target load for every runqueue rq_i represented by a SE se is a fraction of the total load represented by the SE, proportional to its capacity.

$$targetload(rq_i) = load(se) \cdot \frac{capacity(rq_i)}{capacity(se)}$$

If all child SDs are equally sized, this is a simple average.

In order to reach the targeted load for a runqueue, load must be shifted either towards or away from it – alternatively one could influence other parts of the equation, such influencing the overall load of the encompassing SE without touching the runqueue in question. There are several possibilities to achieve different effects and combinations thereof depending on what is allowed by constraints:

1. Balance runqueues represented by a SE.

2. Reduce or increase internal imbalances of SEs within represented runqueues.

3. Reduce or increase weight of the SE in question.

The simplest case is a migration of a task or a group of tasks between runqueues of SDs of equal width: dequeue an SE from one runqueue and enqueue it in another – with both runqueues being part of the group of to-be-balanced SDs. If the dequeued SE represents a part of a coscheduled set with multiple SEs, then the target runqueue may not already have another SE of the set in question. Then, this is a pure load movement between the two runqueues in question without affecting anything else. From here, there are several variations. If one or both of the runqueues are only reachable by traversing the SD hierarchy downwards (via different links), then the migration will also affect the balance within passed SDs. Care should be taken, that such a migration does improve the overall situation. If a nested set has (or may have) multiple SEs, then load can be shifted partially from one of these split SE to another – potentially creating further split SEs or merging existing ones. If the SD width required by an SE is not fixed, i. e., the coscheduled set can be resized, then load movement may also occur between SDs of different widths. Depending on the style of fairness for this set (cf. Section 6.1.3), the weight of the moved SE might change: the same-CPU-time fairness is neutral; the same-system-time fairness makes SEs lighter when enqueuing them in smaller SDs. As a special case, if load is moved up- or downwards the SD hierarchy, a runqueue can be made lighter or heavier without influencing its siblings.

The design itself has no requirements for concrete algorithms for load movement. Though, there are some more guidelines to follow than just simply balancing every SD on its own. Specifically, the load balancing – as it has been described – is effective only in the presence of time constraints. If a set has just placement constraints, the placement will adhere to the constraint, but the load it not necessarily balanced. This may or may not be an issue for a particular use case. Thus, on a case by case basis, load balancing could handle a set as if there was just one horizontal link into the parent instead of many, or load balancing could avoid to let allowed CPUs to go “idle” with respect to the set. Additionally, it will not always be possible to achieve a perfect balance, which might require some additional tweaking of the balancing. Some options are discussed in the following two sections.

6.4 Fragmentation avoidance

The SD hierarchy is not only used to realize the different scheduling constraints, but also to manage fragmentation: The imposed structure is used to enforce

good mappings and to easily find work for CPUs if the current coscheduled set cannot keep them busy.

6.4.1 Idle selection

A coscheduled set consists of a set of interconnected SDs. This set is assumed to follow a global structure – for example that of the system topology. In that scenario, applications have very little options in their degree of parallelism: core, socket, system. When some CPUs cannot be kept busy, this leads to fragmentation. Presently, in order to address this, it would require enough CPUs going idle, so that the whole application can be shifted to one of the smaller child SDs.

However, due to each coscheduled set following the same global structure, this means in particular, that the parent set has similar SDs as the current set. This allows idle CPUs to pick up work without violating any coscheduling guarantees: Whenever an SD within a group is idle, the scheduling decision proceeds within the identical SD in the set's parent. This way, the current and the parent group are still coscheduled. That is, the SDs of the parent set are effectively scheduled with idle priority within their children forming *reverse horizontal connections* within the SD hierarchy. (The extra time some SDs are executed due to this idle selection, leads to a skew in fairness and their weights are no longer proportional to the received CPU time. This is addressed with a global solution in the next section.)

To make this idle selection more efficient, care must be taken during load balancing to free fewer larger SDs instead of more smaller SDs. With only small SDs, larger applications are prevented from being executed and there might not be enough small ones to fill the holes. Thus, the balancing code must track the number runnable tasks within the SD hierarchy, so that an informed decision can be made to migrate tasks from one half-filled SD to another once the combined load can be sustained by a single SDs. Something similar is actually already done by current operating system schedulers – only in the context of power saving: when all load fits onto one processor, it can be consolidated, so that the second processor can be put into a deeper sleep state, conserving energy.

6.4.2 Asymmetric SD shapes

For cases, which cannot be handled by the idle selection alone, it is also possible to, e. g., realize a 5:3 split of computational resources by creating appropriate SDs and let them co-exists with the "regular" SDs in a coscheduled set. Having multiple hierarchies available at the same time is also required to support different kinds of placement constraints simultaneously.

To keep the SD hierarchy from extensive growth due to a combinatorial explosion and to keep load balancing and idle selection efficient, these different hierarchies must not fan out into their own leaf SDs. Instead the hierarchies converge to the same set of leaf SDs. This can either be achieved by enqueuing

an SD in multiple parent SDs by splitting its load across its parents as described in Section 6.1.3, or by a set-internal application of the idle selection itself – or a combination of both.

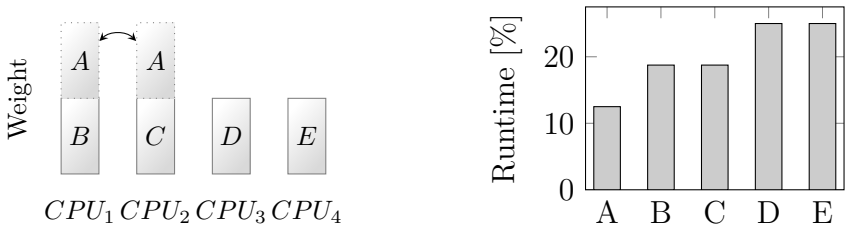
The difference between explicitly realizing a certain split, whether it is into a separate hierarchy or joining multiple hierarchies again, and relying on the idle selection, is that the former is more suited towards stable situations due to setup costs, while the latter handles dynamic situations just fine but is not as versatile.

6.5 Tracking of CPU time

Usually, it is impossible to completely avoid all load imbalances. This might cause SEs to receive more or less CPU time than intended. Especially, in moments of changing load, some SEs might gain an advantage until the load balancing mechanism is able to almost balance everything out again. Even then, these small imbalances accumulate over time. Until now, the presented mechanisms do not handle this. That is, a SE that – for whatever reason – gains an advantage over other SEs, will not be automatically penalized in the future. One possibility is to occasionally shift the imbalance load to even everything out – at least probabilistically. But this is on the one hand not that exact, and on the other hand problematic in the hierarchical scenario, where load can also be changed in width.

Additionally, idle scheduling and enqueuing on SE multiple times introduce new conceptual imbalances: some runqueues below an aggregating SE get more CPU time than others. Hence, balancing weight equally is no longer adequate. While this could be addressed by giving runqueues different processing capacities and balancing load proportionally to this (essentially a variant of asymmetric multiprocessor scheduling), the fact remains, that the amount of extra capacity is mostly just guessed.

To address these issues altogether, it is possible to track the received CPU time of a SE, compare it to the expected CPU time, and design the scheduling and balancing algorithms around that. (For example, older versions of VMware do this on a per-VM basis, cf. Section 4.4.) In this thesis, a different approach is suggested, which works well together with the proportional-share scheduling and balancing in the hierarchical setup. Simply put, the weight of a SE is dynamically adapted when it is ahead or behind its expected time. The more behind a SE is, the heavier it gets; the more ahead, the lighter. Within a single runqueue this has the advantage, that runtime differences are evened out slowly. For example, if a SE got ahead for some reason, it will still be executed – albeit a bit slower – instead of being forced to sit out until the others catch up (and possibly causing a noticeable lack of response for the user). In the context of multiple runqueues, this concept causes SEs on runqueues, which get less runtime than appropriate, to get heavier over time, until the increasing weight imbalance



(a) Weight distribution. (b) Resulting CPU time distribution.

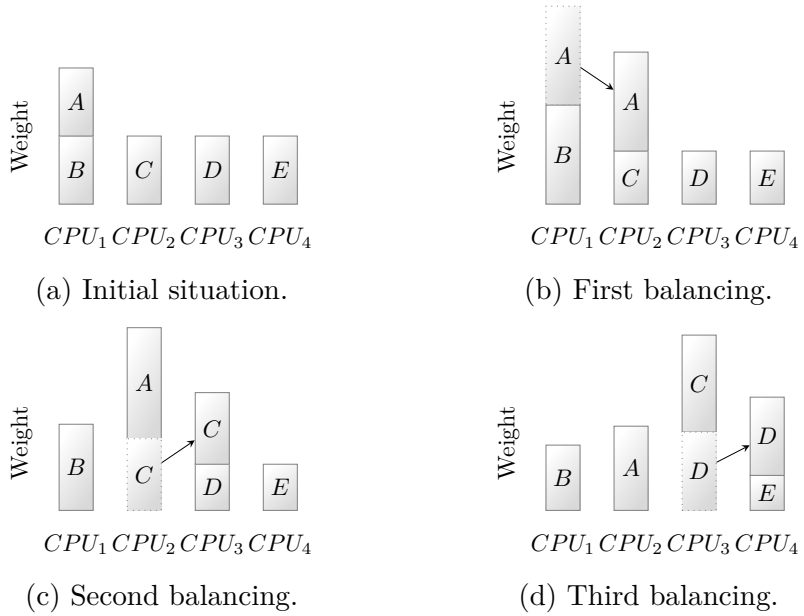
Figure 6.3: Worst case situation for a simple periodic rebalancing algorithm: task A is bounced between two CPUs.

triggers a rebalancing between runqueues. Compared to a simple periodic shift of imbalance without dynamic weights, this variant always migrates the "correct" SEs. Another effect is, that heavier queues are less attractive for new SEs.

As an example, consider five similar tasks on a quad-core system. One of the available CPUs will have to execute two tasks, which then do not make as much progress as the other three tasks. With just a simple periodic rebalancing, it could be that always the same task is bounced between two CPUs, resulting in one really slow task, two sometimes slow tasks, and two fast tasks as depicted in Figure 6.3. With weight adjustments, it is not necessary to support periodic rebalancing at all. Instead, it is sufficient to balance tasks only, if the overall balance is actually improved (and the improvement is above a certain threshold to prevent rapid re-balancing). In the given example, which is illustrated with values obtained from a simulator run in Figure 6.4, the two tasks on the overloaded CPU will get heavier over time (while the others get lighter). A point will come, where moving one of the heavier tasks off its CPU makes sense, as the overall balance is improved. The receiving CPU now has two tasks, a heavy and a light one. This will cause the heavy one to catch up to the light tasks until their weights even out and each task on this CPU receives again 50% of CPU time. While this is going on, globally both of these tasks will get heavier compared to the tasks running on the other three CPUs – again until it makes sense to move one of the two tasks to another CPU. The CPU that had two tasks originally will not be the receiving CPU this time, because its task is still heavier compared to the tasks on the other two CPUs.

In the context of the idle selection from Section 6.4.1, the dynamic weight adjustment causes tasks that receive extra CPU time to get lighter. At some point, the runqueue would attract additional load, either during balancing or during task creation. As a special case, when there is no additional load that can be pulled, the load balancer should be able to exchange a light SE with a heavy SE so that another SE can profit from the additional capacity of the runqueue.

To realize these dynamic weight adjustments, the CPU time received by each SE is tracked. To make the times comparable between SEs of differing base weights and hierarchy levels, the times are normalized to the default task



(a) Initial situation. (b) First balancing.

(c) Second balancing. (d) Third balancing.

Figure 6.4: Same situation as in Figure 6.3 with dynamic weight adaptation. Whenever an imbalance threshold is crossed, the best fitting task is migrated from the most to the least loaded CPU. The fourth balancing would move E onto CPU_1; we are roughly back to the initial situation and the cycle repeats.

weight. That is, received CPU time is tracked one-to-one for tasks with the default weight, while time for a task of half that weight progresses twice has fast. Hence, when all SEs have received their proportional share of some time interval, they will all have experienced the same increment for their normalized CPU time.

Definition 24 (Normalized SE CPU time)**.**

$$ntime(se) \mathrel{+}= \Delta time(se) \cdot \frac{w_0 \cdot capacity(se)}{weight(se)}$$

By comparing individual SE times with the average time of a group of SEs, e. g., a runqueue, it is possible to determine whether a SE is behind or ahead.

Definition 25 (Normalized runqueue CPU time)**.** The normalized time of a runqueue (or any other type of group) is the weighted average of its SEs.

$$ntime(rq) = \sum_{se_i \in rq} ntime(se_i) \cdot \frac{weight(se_i)}{weight(rq)}$$

Usually, it is not necessary to calculate this value explicitly, because it can be substituted by the normalized time of the enclosing group/aggregating SE, which is already available via the regular SE CPU time accounting.

Definition 26 (Dynamic SE weights)**.**

$$dynweight(se) = weight(se) \cdot 2^{f \cdot (ntime(rq) - ntime(se))}$$

Dynamic weights make a SE heavier or lighter depending on whether it is behind or ahead, respectively. The larger the difference to the target time, the more the dynamic weight will deviate from the original weight. The aggressiveness of the weight adjustment can be controlled with the parameter f. For example, when set to $\frac{1}{60s}$ it means that a tasks that deviates by a minute of normalized runtime from the average, will execute twice (or half) as fast as usual to make up for it. The unbounded growth ensures an eventual rebalancing in cases, where a small change is not able to shift the imbalance for the necessary amount. If the dynamic weight goes towards zero, the SE makes too much progress and it indicates that it might be necessary to artificially stop the SE for a moment.

These dynamic weights are used to calculate the share of a SE and for load balancing, but not for the actual accounting of received CPU time. The given definition has the advantage, that several actions within the SD hierarchy can be short-circuited, because $2^{c-b} \cdot 2^{b-a} = 2^{c-a}$. For example, for load balancing within a coscheduled set made up of a multi-level SD hierarchy, dynamic weights of individual SEs can be derived directly, without having to stop at intermediate levels to determine their contribution to the dynamic weight. To prevent new SEs or SEs woken after a long sleep from "catching up" for extended periods of time, their normalized time is initialized (or reinitialized) appropriately. Newly forked tasks can be initialized with the time of the currently running task. Woken tasks keep their normalized time, unless it is considered too far behind in which case it is forwarded to a more considerate time. If it is clear, which other task caused the wakeup, it is also possible to reflect the causality by copying the corresponding time if it is larger.

6.6 Chapter Notes

A less flexible version of the basic structure in Section 6.1 was previously published by myself in [84]: SDs were fixed at the time, the SD hierarchy homogeneous, and fairness across hierarchy levels had not been given that much thought. In a sense, this publication has been the proof-of-concept that pursuing a thesis in this direction is worthwhile.

Chapter 7

Integrating TACO

After the presentation of the coscheduler design in previous chapter, this chapter focuses on the application of said design. A method is described, which converts an existing "regular" multicore scheduler into a scheduler capable of coscheduling. Behavior and features of the existing scheduler are preserved, enabling a smooth transition towards coscheduling-enabled systems.

This chapter starts with the requirements for a successful conversion in Section 7.1 and a description of the conversion method itself in Section 7.2. Afterwards the method is applied to Linux in Section 7.3, which results in the coscheduler that is used throughout the remaining thesis. The chapter closes with a study of FreeBSD in Section 7.4 and how the method would be applied there.

7.1 Requirements for Integration

In order to add coscheduling functionality to existing schedulers while preserving their existing set of features, an existing scheduler is dissected into its building blocks. These building blocks are then rearranged and put back together. Depending on the original scheduler and the desired coscheduling features, this process can be more or less complex. In particular, it is a very scheduler-specific process.

The scheduler in question needs to fulfill certain requirements, so that the process can be applied easily. Throughout this thesis, this is colloquially expressed by requiring a *general purpose multicore scheduler* as a base to work from, which not only conveys an idea of the targeted application domain but also sets some expectations on scheduler properties, that can be taken for granted. The term "multicore" indicates the need for a parallel system and a corresponding scheduler; on non-parallel systems there is no coscheduling possible. Technically, the system does not have to use (just) multiple cores to achieve parallelism, but it currently the most widespread method. The "general purpose" part implies an absence of structural specialization one might find in single purpose schedulers, such as fixed process lists or an absence of preemption in some embedded/real-time systems.

The original scheduler is considered to be assembled from the following building blocks:

- *scheduling entities*, which represent either tasks or groups of tasks;
- *runqueues*, which hold scheduling entities;
- *functions operating on a single runqueue*, such as enqueuing and dequeuing scheduling entities and determining the next scheduling entity to execute;
- *functions operating on multiple runqueues*, such as load balancing functions and task placement.

In addition to these structural requirements, the original scheduler has to fulfill two additional requirements, so that adequate coscheduling can be orchestrated: Firstly, the original scheduler must support preemption. Without preemption there would be no mean to enforce any kind of useful coscheduling. Secondly, the original scheduler should support some kind of weight proportional scheduling. Otherwise it will not be possible to keep CPU time ratios of the various tasks identical to those of the original scheduler – or only in a very limited fashion.

There are no other limitations on the design of the original scheduler. However, because all building blocks are reused during the conversion, drawbacks of the original scheduler will resurface in the converted scheduler. For example, it does not matter for the conversion, whether the original scheduler utilizes a global runqueue or distributed runqueues in some way, like one runqueue per CPU. In case of a global runqueue, the resulting scheduler will still utilize global runqueues; it will not suddenly become scalable. Similarly, if the original scheduling algorithm is susceptible to certain attacks, they will still work after the conversion.

7.2 Conversion of Non-coscheduling Schedulers

The process of augmenting a scheduler with support for topology-aware coscheduling can be split into five consecutive steps. Each step has to be applied with the specific scheduler in mind that is being converted. There is no one-fits-all implementation. After each step, the scheduler is still fully operational and retains its properties. At the end, coscheduling is supported. Note, that some locking and performance aspects are discussed only afterwards.

Step 1: Support a task grouping mechanism. Coscheduled sets are groups of tasks. Hence, infrastructure is needed to manage groups of tasks. Ideally, the operating system already supports a container mechanism that is generic enough. Note, that the one container for tasks, which is likely to exist in an operating system – namely the concept of a process – does not provide

the necessary flexibility: it is too tightly coupled to address spaces to allow a wide variety of coscheduling use cases. For full flexibility, a container concept is needed that is orthogonal to processes.

Step 2: Runqueue generalization. If not already supported, the runqueue data structure is generalized, so that runqueues can be allocated and freed on demand (for later dynamic creation of coscheduled sets) and are able to hold different types of *scheduling entities* – the basic type of a scheduling entity being a task. In case the runqueues are protected by locks, the locks need to be decoupled from the runqueue data structure itself.

Step 3: Hierarchical organization of runqueues. The runqueues are reorganized hierarchically. The "normal" runqueues for tasks become the bottom layer, and runqueues representing subsets of these runqueues are added on top – each runqueue representing a synchronization domain. A new type of scheduling entity is added – the *synchronization domain SE* (SD-SE) – which represents the runqueues below it in the hierarchy. Task enqueuing/dequeuing and task selection are adapted to work along the hierarchy. If desired, it is possible to apply the hierarchical reorganization to only a subset of the scheduling classes/priorities supported by the operating system. This allows, for example, to handle kernel activities outside the scope of the coscheduler.

Step 4: Adaptation of functions operating on multiple runqueues. With coscheduled sets there will be multiple sets of runqueues. The load balancing mechanism is adapted, so that it is able to work within only a certain set of runqueues. This prevents tasks from switching between sets unexpectedly. Similar adaptations are done for other functions that operate on multiple runqueues, such as the selection of a runqueue for newly created or woken tasks.

Step 5: Support for coscheduled sets. Infrastructure to create and destroy additional sets of hierarchical runqueues is added – each additional set represents a coscheduled set. This is coupled to the task group mechanism. Knobs are added to be able to configure the behavior of the coscheduled set. While some of these will be coscheduling-specific, others will resemble aspects that were previously only available for tasks but apply to coscheduled sets as well – weight and CPU affinity for example.

The load balancer is not only applied to the runqueues in the root set, but to each coscheduled set individually. For on-demand invocations, like idle or wake-up balancing, this is a straight-forward application. For periodic invocations, one option is to keep track of the period per set and base the period on CPU time instead of wall-clock time. This would avoid balancing sets, which are currently not running anyway, while not deviating from the behavior of the original balancer.

At this point, the newly integrated coscheduling functionality is already fully functional. But some things can still be improved to work better with the newly introduced hierarchical runqueue organization. For example, one could apply the original load balancer on each level of runqueues within a coscheduled set. However, this would not be able to address imbalances between different levels, where – for example – one half of the bottom level runqueues have to shoulder a higher load, so that they offset the extra load caused by small coscheduled sets on the other half. If the original load balancer supports asymmetric load distributions, one could address this by propagating the remaining imbalance of one level down to the next level. (Note, that idle balancing will also take care of some scenarios without doing anything special.) This leaves only load balancing by resizing of coscheduled sets, for which there is nothing similar in traditional schedulers.

The hierarchical organization of runqueues requires careful thought to avoid deadlocks in the locking scheme, if the original runqueues use fine-grained locking, e. g., one lock per runqueue. Applying a multi-granularity locking scheme would solve the issue without too much thought. However, it comes with additional overhead on every operation on any runqueue. Instead, it is possible to stay close to the original locking scheme, grabbing locks only for those runqueues/levels that are needed for the current operation. For this, the per-runqueue locks are replaced with per-SD locks with all runqueues for a particular SD referencing that lock. Additionally, lower-level locks are always acquired before those on higher levels. This way, one of the necessary conditions for a deadlock, namely circular wait, is avoided. This works for enqueuing and dequeuing, which can bubble upwards in the hierarchy via lock chaining, and for the scheduling decision, for which the master has to get all locks up to its personal root before it starts picking scheduling entities downwards within the hierarchy. If load balancing (or something else) requires to get multiple locks within the same level of the hierarchy, the solution of the base operating system is reused. For example, Linux acquires them by ascending memory addresses.

Another solution – which is also required when the base operating system uses lock-free runqueues – is to not require atomicity on certain runqueue operations, so that it is not necessary to hold multiple locks at the same time and runqueues can be handled one at a time. But that means that, e. g., task picking may traverse downwards into a coscheduled set, which gets dequeued concurrently. The employed algorithms must be able to handle this and other cases. Hybrid approaches are also possible, where for example dequeuing is done lazily. That is, dequeuing never bubbles the hierarchy upwards. Instead, the actual dequeuing is done when empty coscheduling entities are encountered during task selection.

Figure 7.1: Scheduling classes in Linux. They are processed in order when selecting a task: the first task found is executed.

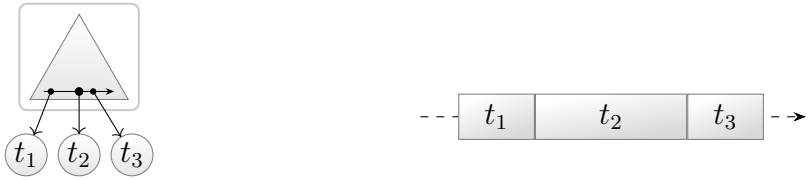
7.3 Coscheduling in Linux

To demonstrate the feasibility of the proposed coscheduler design and integration method, an integration into the Linux Completely Fair Scheduler (CFS) [83] was done. The result – a Linux 3.8 kernel with a CFS capable of coscheduling – is evaluated and used in later chapters. This section describes its non-intrusive integration. This starts with an introduction into the building blocks of the Linux scheduler in Section 7.3.1 and continues with their reutilization to realize coscheduling in Section 7.3.2. Section 7.3.3 covers the adaptation of the load balancer separately. Finally, Section 7.3.4 discusses a few decisions done during development and outline a potential future direction.

7.3.1 Linux Scheduler Basics

The last overhaul of the Linux scheduler happened with Linux 2.6.23 in late 2007. Since then, the Linux scheduler subsystem is somewhat modular with statically prioritized scheduling classes and per-CPU runqueues. Technically, there are currently five scheduling classes (in decreasing priority): stop, deadline (since Linux 3.14), realtime, fair, and idle (Fig. 7.1). The stop class is only used internally to realize, e. g., active balancing. The deadline and realtime classes are for tasks with special needs regarding CPU time that are executed before normal tasks – these two classes are considered out of scope for this implementation. Normal tasks are placed in the fair class, which is handled by the Completely Fair Scheduler. Only when there is no task to execute, the idle task in the idle class is run.

The Completely Fair Scheduler (CFS) has its name for its exact accounting of CPU time that does not use the concept of (fixed) time slices. Instead, for every task a virtual runtime is tracked and updated whenever that task is executed. The runqueue on each CPU is a red black tree that is ordered by the tasks' virtual runtime. Anytime a scheduling decision has to be done, the leftmost task (with the smallest virtual runtime) is selected. This prevents individual tasks from getting an unfair advantage over other tasks. New tasks get a "current" virtual runtime assigned; sleeping tasks keep their virtual runtime to some extend. This way, I/O-intensive tasks are usually leftmost enough within the tree after being woken to facilitate a preemption of the currently executing task. After being selected each task is allowed to execute for some time unless something more important comes up. This "time slice" is calculated dynamically and depends on several factors – among them current load, importance, and the number of CPUs in the system. CFS does not use absolute priorities (that is what the realtime



(a) CFS runqueue with tasks ordered by their virtual runtime, t_2 has twice the weight of t_1 and t_3.

(b) Resulting schedule, repeating itself until a change occurs.

Figure 7.2: Scheduling with CFS. Tasks receive CPU time proportional to their weight. Execution time (scaled according to weight) increments virtual runtime; task with least virtual runtime gets executed.

class is for). Instead CFS realizes a proportional share scheduling, where each task gets CPU time proportionally to its weight. That is, for more important tasks (virtual run-)time progresses slower compared to other tasks. The weight is controlled via nice-values.

This is illustrated for a single CPU in Fig. 7.2. Figure 7.2a shows a CFS runqueue with three tasks ordered by their virtual runtime, the enqueue dot representing their weight. The leftmost task is t_1, hence it will be executed first. After being executed for some time, it is reinserted into the runqueue, usually to the right, and execution continues with the next task. Task t_2 weights twice as much as the other tasks, so it is executed twice as long. However, since its virtual time is slowed down by a factor of two due to the weight, the virtual runtime of t_2 is incremented by the same amount as the other tasks. So unless the overall situation changes, we see a schedule similar to the one shown in Fig. 7.2b. When a sleeping task, say t_4, is woken while t_1 is running, t_4 is inserted into the runqueue and a preemption check is done by comparing its position to the updated position of t_1. If t_4 is enough towards the left, a preemption is triggered. However, which task is executed next also depends on the relative placement of t_4 to t_2 in this case. (All checks involve some thresholds to avoid unfavorable switches.)

With Linux 2.6.24 a generic mechanism to handle groups of tasks was integrated. These *control groups (cgroups)* are given purpose by attaching one or more cgroup subsystems to them. The scheduler brings it own cgroup subsystem, which allows realizing fairness not between individual tasks but between groups of tasks. This is done by representing the group by a *scheduling entity (SE)* with its own user-controlled weight and virtual runtime. This SE is enqueued in the runqueue in place of the tasks, when there is at least one runnable task. To further ensure fairness between tasks in a group, these tasks are managed in a runqueue of their own. This structure also allows arbitrary nesting of task groups. Figure 7.3 shows an example of nested task groups on a uniprocessor system. All task groups have the default weight, which gives a task group the same importance as a task. Thus, each of the three SEs in the system runqueue

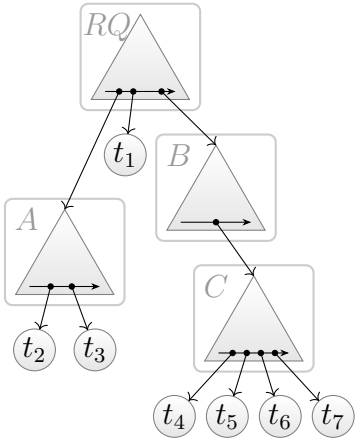


Figure 7.3: Nesting of task groups. No matter how much weight is accumulated, the parent only sees a fixed weight for the whole group.

will receive one third of the available CPU time overall. Within task group A, CPU time will be distributed fair between t_2 and t_3, i. e., each gets one sixth of the overall CPU time. This requires also a modified task selection algorithm: Starting from the system runqueue, the scheduler now repeatedly selects the leftmost entity of a runqueue until it finds a task, which is then executed. When necessary, e. g., when the calculated time slice for the task has expired, execution time for now gets accounted at all SEs along the previously traversed path. The next scheduling decision is again started from the top. In Fig. 7.3 task t_2 is selected first via task group A. After finishing its slice, it is reinserted right of t_3. Additionally, the SE of task group A is updated and moved accordingly – probably somewhere to the right of t_1, maybe even right of the SE of task group B. Thus, the next task is t_1 followed by either t_3 or t_4, depending on the actual values.

Handling of task groups in the multicore case is a bit more involved, because of the per-CPU runqueues. Task groups follow this design decision and have one runqueue per CPU allowing tasks of a group to be distributed over multiple CPUs. This also requires multiple representative SEs instead of just one. Linux solves the issue of fairness by distributing the user-controlled weight of a task group between all of its representative SEs. This splitting is done proportionally to the load realized by the task group on the corresponding CPU, so that each task within the task group still receives its fair share no matter the actual distribution of tasks. Figure 7.4 shows a task group in a quad-core system with four tasks of equal weight. The load distribution within the group is 3:1:0:0. The task group itself has same weight as task t_1, which is split between the two SEs representing the non-empty queues of the task group. With four tasks in the group, each task should receive one quarter of the CPU time t_1 receives. Thus, the SE representing the queue with three tasks get three times the weight of the other SE. Of course, if we would have this situation in a real system, the load

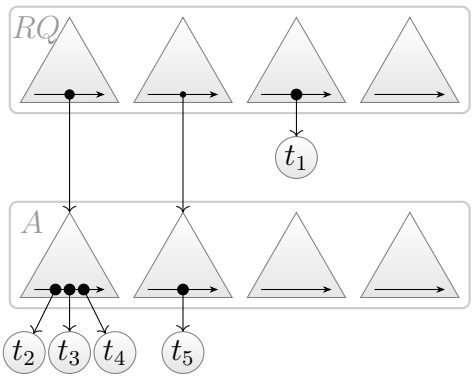


Figure 7.4: Task group handling on a quad-core system. Each CPU has its own set of runqueues: one system runqueue and one per task group. The weight of a task group is fixed and is distributed across its SEs reflecting the internal load distribution.

balancer would do something here. However, the balancer ignores task group membership and only strives to balance load within the system runqueues. In this example, this would mean to migrate t_2, t_3, or t_4 towards the idle CPU, which already yields the best possible result – still giving each task more CPU time than strictly necessary.

The load balancer tries to distribute and redistribute work evenly across the CPUs. For scalability, balancing between physically close CPUs is done more often than between CPUs further apart. Internally, this is realized by defining – per CPU – a set of successively larger system partitions called *scheduling domains (SDs)*. Each SD is partitioned into *scheduling groups (SGs)*. These SDs reflect the system topology and with the exception of large NUMA systems, each SD is used as an SG in the next larger SD. (On systems with many NUMA nodes a distance metric is used to create SDs consisting of increasing numbers of close NUMA nodes). Thus, on a typical multicore system there is a SD a) for a core (in case of SMT), b) for a processor, and c) for the system. Figure 7.5 shows this for a dual quad-core system, which is used later in further examples. Except for NUMA level SDs, the higher level SDs are shared between participating CPUs.

The load balancer always works with an SD as input; with smaller ones getting processed more often. Within an SD, the balancer tries to achieve a balance between SGs, so that all SGs have the same load. The load distribution within a SG is ignored, as this will be addressed when the corresponding SD is processed at another time. The balancers starts by collecting and aggregating load information from each runqueue within a SG. When there is an imbalance between SGs, the balancer tries to migrate an adequate amount of tasks from the busiest SG to the least loaded SG, so that the overall situation is improved. As a first guess, the busiest runqueue within the busiest SG is selected as source and the idlest runqueue within the least loaded SG as destination. Though, there is logic to handle cases, when tasks can not be migrated for some reason

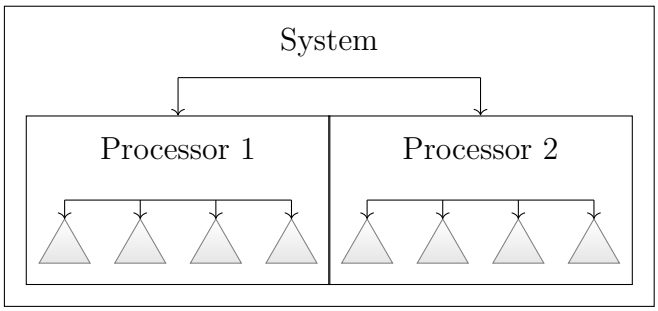


Figure 7.5: Scheduling domains on a dual quad-core system. The load balancer strives to achieve balance between the indicated scheduling groups by migrating load from one group to another.

(e. g., pinned tasks), or when enough capacity is available so that, for instance, a whole processor can be put into sleep mode. The balancer only works with tasks. In case of task groups, the balancer considers these tasks on base of their hierarchical load. Going back to Fig. 7.4, the balancer would consider t_2 to t_5 to have a quarter of the weight of t_1, each.

Over time, this achieves a system-wide balance. However, transient imbalances may exist and some imbalance is even allowed to cut down the number of migrations. Thus, some tasks may receive more CPU time compared to others than indicated by their weight. While large imbalances are countered by periodic rebalancing (e. g., three tasks on a dual core) to reduce this effect probabilistically, it is not explicitly tracked. In addition to this regular behavior, limited balancing is also done on demand, e. g., when a CPU is about to go idle or a task is about to be created or (in some cases) woken up. This way, some imbalances are not created in the first place.

7.3.2 Scheduled Task Groups

The existing task group mechanism provides an ideal base for coscheduled sets, because they already realize a part of the accounting that is essential for many use cases: fairness at application level. Additionally, the surrounding cgroup system already allows nesting and includes infrastructure to easily set up configuration knobs necessary to configure coscheduled sets. With these mechanisms already in place almost no additional work needs to be done for Step 1 (task grouping) and Step 2 (runqueue generalization) of the conversion method. Only locking and unlocking of runqueues is abstracted, so that lock-accesses are no longer hard-coded. For the other steps, more work is necessary.

The load balancing in Linux is tightly bound to the scheduling domains, which already provide a way to restrict balancing to a subset of CPUs (but not yet to individual task groups). Therefore, aligning the hierarchy of runqueues with the hierarchy of scheduling domains fits the existing load balancer rather nicely. Differently structured hierarchies can then be achieved by influencing

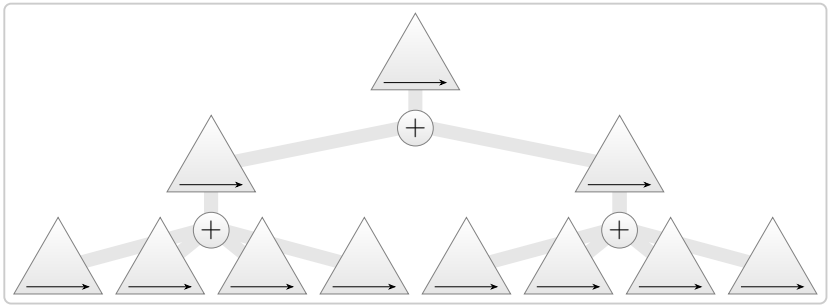


Figure 7.6: Empty runqueue hierarchy on a dual quad-core system aligned with scheduling domains. For every non-leaf runqueue a SD-SE exists, which represents the group of runqueues below it.

the generation of scheduling domains. Only the feature of having individual scheduling domains per NUMA domain needs to be deactivated.

Allocating an additional runqueue for each scheduling domain creates the base runqueue hierarchy. For each new runqueue, there is also a new scheduling entity representing the scheduling group below it. This *synchronization domain SE* (SD-SE) is enqueued, when at least one of the represented runqueues has load. In that, it follows the already existing logic for enqueuing and dequeuing of task group SE, just with a logical OR. This base hierarchy is depicted in Fig. 7.6 for a dual quad-core system, where three additional runqueues are created: two for the processors and one for the whole system. Together with this hierarchy, the modified task selection logic is introduced. Where each CPU started at its own system runqueue before, they now start their selection process at a runqueue that has to be explicitly specified. For one CPU this is preinitialized with the top runqueue in the base hierarchy. Whenever a CPU picks a SD-SE, it continues the selection process in the next smaller SD, which contains the CPU itself. For every other SD represented by the SD-SE, one CPU is selected and is notified via an IPI to begin a task selection of its own starting at the SD's runqueue. This concludes Step 3 (runqueue hierarchy) of the conversion method. (Note, that Step 4 (load balancing) is covered separately in Section 7.3.3.)

For Step 5 (coscheduled sets) of the conversion method, the already existing task groups are extended similarly to Step 3 with hierarchical runqueues and scheduling entities. Together with the SE that comes automatically with each runqueue in a task group, each runqueue can now be represented by one of two scheduling entities: via the SD-SE in the parent runqueue, or via the original task group SE (TG-SE) in the corresponding runqueue in the parent task group. Depending on how a task group is configured, one or the other SE is used. Besides resulting in the ability to realize coscheduled sets of different sizes, it also allows retaining the original task group functionality of group based accounting. We differentiate between *regular task groups (RTGs)* and *scheduled task groups (STGs)*. The former do not make use of intra-group SEs: any nonempty runqueue is represented separately and scheduled independently. Scheduled task

groups usually have exactly one link into the parent task group, presenting a united front to the parent group. This difference is illustrated in Fig. 7.7.

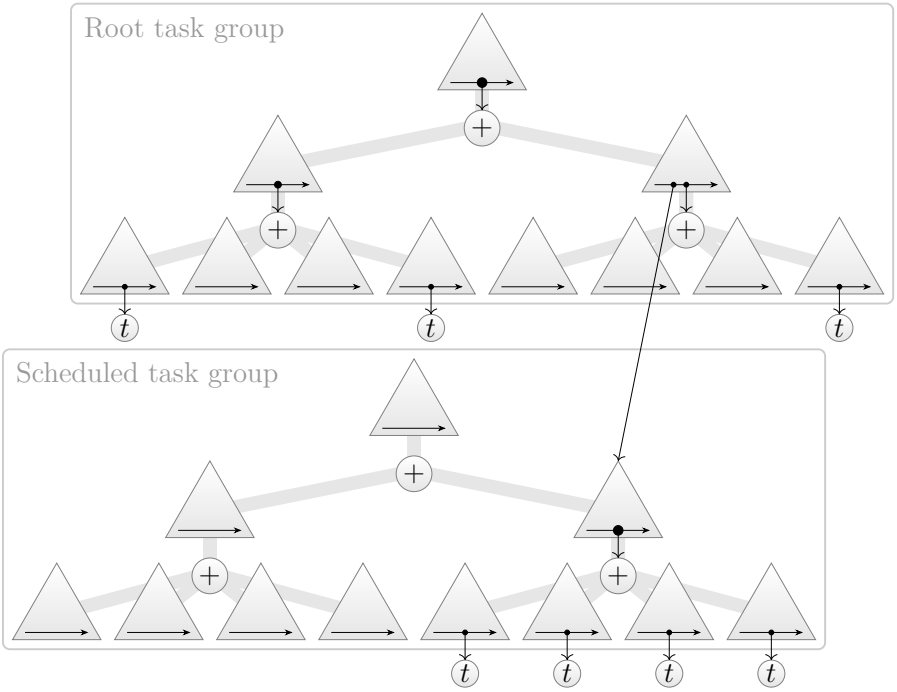
While it may seem a bit excessive to create a whole set of runqueues even for small coscheduled sets, it is necessary for dynamic size reconfigurations to be able to finish without memory allocations. Additionally, it is currently necessary to allow migrations of whole coscheduled sets, which have to be done task by task as Linux is not prepared to suddenly use a runqueue on another CPU than it was created for. Due to this, some hybrid forms of task groups may briefly occur that are neither RTGs nor STGs.

7.3.3 Load Balancing

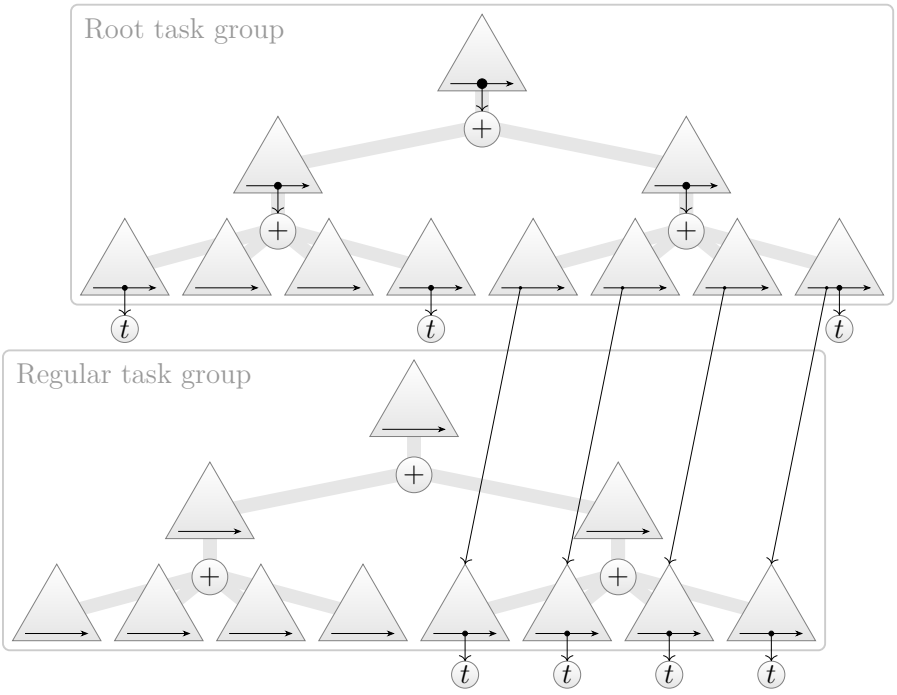
For Step 4 (load balancing) of the conversion method the load balancer is adapted to work with the runqueue hierarchy. The previous "balancing of scheduling groups within a scheduling domain" now translates to "balancing of runqueues represented by a SD-SE". Instead of gathering statistics over a bunch of runqueues, statistics have now to be gathered for a subtree in the hierarchy. This actually simplifies the process a bit, as certain needed statistics are automatically gathered by the hierarchical runqueues. The handling of regular task groups in the hierarchical case still works the same way as before. The only problem here is that the load balancer has actually to be prevented from looking into scheduled tasks groups, as these have to be migrated en-bloc. Just migrating a single task would destroy the STG structure. Instead, the contents of a STG are balanced separately (the base hierarchy is also a STG).

The independent balancing of different STGs is possible, as all CPUs used by a STG operate simultaneously and, thus, get the same amount of CPU time. This independence of STGs makes it possible to only balance STGs that are currently active, which keeps the overhead down. Note, that the active STG does not only depend on what is currently executed but also on the hierarchy level that is currently subject to load balancing. For example, consider a system executing a lot of small STGs; when balancing at a low hierarchy level, this will balance within the currently executed STG, while balancing at a higher hierarchy level will balance the small STGs themselves. When each STG is balanced, the whole system is balanced.

Note, that the correction of certain imbalances within an STG was not addressed. For example, if a coscheduled set does not occupy all CPUs of its synchronization domain, it would be beneficial to free a partition as large as possible. Though it was not done, it is possible to use the already existing balancing mechanism for energy savings, which keeps whole processors idle if other processors have enough capacity to execute all tasks. Similarly, imbalances in lower levels of multi-level STGs may remain undetected as long as they average out on higher levels.



(a) Scheduled task group occupying one processor: only a single runqueue is enqueued in the parent. If the STG were to cover the whole system, the link to the parent would be between the top runqueues instead.



(b) Regular task group: each non-empty runqueue is enqueued individually in the parent. If the regular task group had a scheduled task group as a child, we would see an additional link from the RTG into its parent.

Figure 7.7: Scheduled vs. regular task group. The same task group enqueued in different ways into the parent task group.

7.3.4 Current State

The current implementation has still a few deficiencies with respect to the realization of the full versatility of the design as well as some concurrency issues. Both stem from the fact that code reuse was considered more important than realizing a perfect implementation. This way, the implementation demonstrates the maximum benefits with the least amount of changes. It is currently at a point where it can be published to the broader community without being immediately dismissed for being a too big change or being of no use.

The "missing" changes would have to be introduced later, as their benefit per changed line ratio is rather bad. Though some of these modifications would improve Linux in general and are not solely for the benefit of coscheduling.

In no particular order, these missing modifications/features are:

- **Lazy migrations.** Currently, migrating whole task groups is rather costly in terms of locking. Additionally, it has to be done task by task. If migrations could be carried out lazily, i. e., changing the CPU of a task without having care whether it is currently running or maybe even where it is enqueued, this would allow moving tasks concurrently and maybe even en-bloc. For vanilla Linux, the balancer would operate more in the background than it currently does. Also, the additional logic for active balancing would be greatly simplified.

- **Further extension and simplification of balancing code.** The runqueue hierarchy already holds some of the statistics that are required during load balancing and normally only gathered on demand in vanilla Linux. Consolidating these statistics further would allow balancing without having to traverse everything first – albeit at slightly increased costs during normal operations. Also, some more statistics are needed to be able to address the remaining conceptual imbalances.

- **Non-blocking enqueue checks and lazy dequeuing.** Enqueuing and dequeuing have to move upwards in the hierarchy until they hit a SEs that is already enqueued because of load in a sibling. Currently, this last level has to be locked to do the check. A non-blocking check to provide a fast path for the common case would remove one source of lock contention. Another option in this area would be to do dequeues lazily without walking the hierarchy at all. Instead, empty intermediate runqueues would be dequeued when they are encountered during task selection.

- **Non-blocking preemption checks.** Currently, preemption checks in Linux modify the runqueue even when the result is negative. Thus, requiring getting the lock for a runqueue. This is the other source of lock contention, that could be addressed.

- **Generalized scheduling domain code.** While the scheduling domain hierarchy can already be changed during runtime, it is not intended to be done often. Also, having multiple hierarchies at the same time is currently not possible. Without extensively touching that code, the current coscheduling implementation is limited to a single hierarchy.

As it is, the current implementation extends the existing scheduler by nearly 2,000 lines of code, which is an increase of about 10% in the scheduling subsystem. These 2,000 lines are roughly distributed as follows: 20% are comments, 50% are new functions and data structures, and 30% are additional code in already existing functions. All major code paths within the scheduler remain intact, retaining existing properties. Less than 100 lines of previously existing scheduler code were actually changed (i. e., less than 0.5% of the scheduling subsystem), making coscheduling in Linux an additional feature.

7.4 Coscheduling in FreeBSD

FreeBSD is an Open Source operating system that is used in a wide range of systems. FreeBSD is used in this thesis to demonstrate the applicability of the recipe given in Section 7.2 to other operating systems than Linux. The TACO concept has not been implemented, but the FreeBSD scheduler source code was studied to assess the feasibility. The inner workings of the current FreeBSD scheduler are summarized in Section 7.4.1. An outline of a possible integration of TACO in FreeBSD is then given in Section 7.4.2, followed by a small comparison with the Linux integration in Section 7.4.3.

7.4.1 FreeBSD Scheduler Basics

The FreeBSD scheduler has undergone quite a few changes in the recent years. Starting with a scheduler similar to the one in UNIX, its current state only slightly resembles that. Especially, the scheduling of normal application tasks has been thoroughly changed. There is not much documentation of the current workings of the FreeBSD scheduler. The latest publication [85] describes the version 1.0 of the so called ULE scheduler. It is currently at version 3.0 with some changes on top, and the only documentation – besides the source code itself which does not include motivations – is the blog of its developer Jeff Roberson [86] and the commit logs of the FreeBSD source repository [87]. The following is based on a conglomeration of all these sources.

The current FreeBSD scheduler has independent runqueue structures on each CPU and a simple balancing between them. On each CPU, there are multiple arrays of task queues. Each array realizes a different scheduling category with decreasing priorities: realtime (for interrupt threads, realtime tasks, kernel threads, and interactive user tasks), timeshare (for non-interactive tasks), and idle. Except for the timeshare category, the different queues are used to

Table 7.1: FreeBSD task priorities, types, and runqueues.

Priority	Task type	Runqueue mapping
0..47	Interrupt threads	realtime[0..11]
48..79	Realtime user threads	realtime[12..19]
80..119	Top half kernel threads	realtime[20..29]
120..151	Interactive user threads	realtime[30..37]
152..171	Time sharing user threads (light CPU with negative nice)	timeshare[(x+0..16)%64]
172..203	Time sharing user threads	timeshare[(x+17..45)%64]
204..223	Time sharing user threads (heavy CPU with positive nice)	timeshare[(x+46..63)%64]
224..255	Idle user threads	idle[56..63]

realize different priorities, grouping four priorities per queue and always executing tasks from the first non-empty queue in a round robin (and in some cases FIFO) manner. In earlier versions, this behavior was also used for the timeshare class, where the priority was determined dynamically by the nice level of a task and its interactivity score. With ULE 2.0, this was slightly modified. Now, queues within the timeshare class are processed round robin, and each task – after its time slice ended or it gave of the CPU voluntarily – is placed more or less "ahead" of the currently processed queue, depending on nice level and recent CPU usage. Depending on the interactivity score of a timeshare task, it may also end up getting enqueued in one the realtime queues, so that it gets processed before any non-interactive tasks. The interactivity score is an integer from 0 to 100 and depends on the ratio of voluntary sleep time to consumed CPU time: tasks that voluntarily sleep longer than they consume CPU receive a score less than 50, tasks that compute more than they sleep receive a score higher than 50. With the default settings, interactivity scores (after adjusting them by the tasks' nice values) below 30 are scheduled in the realtime queues. See Table 7.1 for a summary of the relation between task priority, type, and runqueue.

The FreeBSD scheduler does know about the hierarchy of a computer system and uses this knowledge in its balancing algorithms. The computer system is represented as a tree, where each node below the root node corresponds to a shared cache. This tree is used to locate CPUs, that are successively further away, to either pull load from when a CPU is idle or to do a wake-up migration on a more suitable CPU if the cache affinity for lower levels has already expired. In addition to this on-demand balancing, there is a global balancing periodically running on the first CPU, which traverses the full tree to repeatedly move load from the CPU along the highest loaded path to the CPU along the least loaded path.

7.4.2 Design Outline

Compared to Linux, FreeBSD does not provide a lot of infrastructure within its scheduler. That makes some aspects of the coscheduling integration easier and others more complex. On the one hand, there is not as much functionality to retain; on the other hand, there is also not as much functionality to ease the integration. For example, FreeBSD does not have a versatile mechanism to manage groups of tasks (Step 1 of the conversion method), which can be used as a basis for coscheduled sets. There is – of course – the concept of a process, which is a grouping mechanism for tasks. But this is too limiting for most coscheduling use cases. The closest equivalent are FreeBSD's cpusets, where arbitrary processes can be limited to certain CPUs. Each task already contains a reference to a cpuset and the infrastructure for managing cpusets exists. It currently has the limitation, that all threads in a process must belong to the same cpuset. However, according to the documentation in the code, there is no technical reason for this restriction; it is only to keep the interface straight-forward, which would have to be extended anyway, so that the additional configuration knobs for coscheduled sets become available. For Step 2 (runqueue generalization) no infrastructure exists that could be re-used; this has to be done from scratch.

The actual conceptual challenge in FreeBSD is Step 3 of the conversion method: how can FreeBSD's runqueue structure and supporting algorithms be modified, so that a hierarchical runqueue setup is supported? Using a similar argumentation as for Linux, we would like to have coscheduling only for the timeshare class, so that anything more important is able to work outside the scope of the coscheduler (like FreeBSD's interrupt and kernel threads). That is, individual tasks of a coscheduled set may get interrupted for a short time, but without preempting the whole set. This would mean that a CPU looks into its (local) realtime queues for work and then, if it does not find anything, it looks into the (hierarchical) timeshare queues to find a coscheduled set to execute. However, a complication arises, because timeshare tasks may get enqueued in the realtime queues instead of the timeshare queues, when they are considered interactive. If hierarchical queues are only set up for the timeshare queues, but we still apply the interactivity logic to individual tasks, it would lead to the interesting situation, where a task – when it is found to be interactive – will break out of its coscheduled set and get executed on its own in the non-hierarchical realtime queues. This would be unacceptable behavior for many coscheduling use cases; but we do not want to lose this feature of the FreeBSD scheduler either, just because of the coscheduler integration – it would violate the rationale of non-intrusiveness.

To properly integrate the interactiveness logic with coscheduling, interactive tasks must not leave the coscheduled set – their handling must become hierarchical as well. That said, in order to not upset FreeBSD's kernel tasks, the "normal" realtime queues have to be retained. That is, a hierarchical variant of

the realtime queues is maintained in addition to the original realtime queues and (hierarchical) timeshare queues. The new set of hierarchical queues can not only hold interactive tasks within a coscheduled sets, but it also allows classifying whole sets as interactive. Having both variants at the same time also opens the opportunity to support coscheduling of user-realtime tasks if desired (or maybe even mixtures of realtime and timeshare tasks). Also, it can be made a per-set runtime configuration, whether the aforementioned breaking out of coscheduled sets is allowed for interactive tasks (after considering the ramifications of this decision on other coscheduled sets). The interactivity of a coscheduled set can be calculated from the sleep- and runtime of the SE of the coscheduled set itself. The definition is straight-forward (being the same as for a task) and emphasizes the handling of a set as an entity: a set would only be interactive when its tasks sleep and wake simultaneously. Individually interactive but otherwise unsynchronized tasks would lead to a coscheduled set that is seldom sleeping and would be classified as non-interactive, which may not be what is desired when multiple (independent) tasks are represented by an aggregating SE. Here, it may be more appropriate for an aggregated SE to inherit the the priority of its most interactive child, if any. This will schedule interactive tasks in a similar manner to an unmodified FreeBSD, no matter how tasks are aggregated and whether they compete with coscheduled sets. Just like an unmodified FreeBSD, this scheme is susceptible to timing attacks, where the system is flooded with interactive tasks (see, e. g., [88]).

FreeBSD's way of scheduling has no direct notion of fairness. While this is not a problem, when tasks have to compete with other tasks for CPU time (or a coscheduled set with another coscheduled set) as we desire exactly FreeBSD's typical behavior in that case, it makes the CPU time distribution difficult, when a task is ranked against a set. Consider, for example, three tasks that behave identical from a scheduler perspective, i. e., same priority, same interactivity (or non-interactivity). If they were placed in the same runqueue, they would receive the same amount of CPU time. The added hierarchical aspect should not change that, even if two of these tasks have been aggregated and are now represented by one SE which is in the same runqueue as the SE of the third task: the aggregated SE should receive twice the amount of CPU time. Similar observations hold for non-identical tasks; arbitrary grouping should not change the asymptotically received CPU time compared to an unmodified FreeBSD. Given that timeshare tasks are scheduled differently, whether they are classified as interactive or not, we can look at one case at a time.

Conceptually, this problem does not exist for interactive tasks, because they cannot stay interactive indefinitely without going to sleep regularly, keeping their sleep-/runtime ratio this way. In the example above, the aggregated SE with two of the interactive tasks would simply stay interactive until both tasks have gone to sleep or lost their interactivity. However, if the runqueue for a specific interactivity score is saturated with always re-awaking tasks, aggregated SEs will not receive an appropriate amount of CPU time. If this is considered an

issue, a mechanism to adjust the received CPU time is needed – for example time slice length manipulation or temporary priority adjustments.

Within the timeshare queues, task priorities end up manipulating the selection frequency of tasks; time slice length – while depended on the number of tasks in a runqueue – is not influenced by the priority. With the current FreeBSD, the most important task in a runqueue can be picked about 64 times more often than the least important task. Thus, we can assign each priority a weight, so that the most important timeshare queue priority has 64 times the weight of the least important one. Then, we can aggregate weights (as described in Chapter 6) and convert the resulting weight back to a priority for the aggregated SE. Unfortunately, the aggregated weight will run out of the supported range very quickly, requiring a scaling mechanism. FreeBSD already does some scaling by mapping the 72 timeshare priorities (152 to 223) onto 64 queues, when calculating how many runqueues ahead from the current position the task needs to be enqueued: $index = \frac{64}{72} \cdot (priority - 152)$. This is adapted for larger ranges as follows.

Definition 27 (Priority to weight conversion)**.** For non-interactive timeshare tasks, the priority is converted to a weight with a simple subtraction:

$$weight = 224 - priority$$

Definition 28 (Weight to runqueue index)**.** A (possibly aggregated) weight of a SE is converted to a runqueue index by scaling it according to a dynamically determined reference weight:

$$index = 64 \cdot \left(1 - \frac{weight}{weight_{ref}}\right)$$

The reference weight is dynamically adapted depending on the weight values represented by the timeshare runqueues, ensuring that SEs are well spread out across the runqueues and still receive their proper quota of CPU time. For backwards compatibility with FreeBSD, it is at least 72. That is, if there is no aggregation, the runqueue index selection is identical to an unmodified FreeBSD. When a heavier SE is encountered during enqueuing, the reference weight is increased to this value. This scales up the supported range and places lighter SEs correctly from that point forward again. Because lighter SEs are placed further ahead, the reference weight cannot simply be the maximum of all enqueued SE weights. Doing so would penalize recently enqueued SEs, as SEs of the same weight will likely be placed in front of them after a heavy SE has been dequeued. Instead, the reference weight decays linearly with each processed runqueue, so that it would reach zero when all 64 runqueues have been processed once. On the one hand, this ensures that SEs, that do not increase the reference weight, are placed correctly with respect to already enqueued SEs. On the other hand,

the reference weight will be always as least as heavy as any already enqueued SE without, for example, having to track the maximum explicitly.

Definition 29 (Timeshare reference weight)**.** The reference weight $weight_{ref}$ is adapted for every enqueue operation of a SE with weight $weight$. Let t be the number of runqueues that were processed since the last actual increase of the reference weight to a value $weight_{orig}$. Then, the reference weight is calculated as follows:

$$weight_{ref} = \max\left\{\left(1 - \frac{t}{64}\right) \cdot weight_{orig}, weight, 72\right\}$$

Locking in FreeBSD is fine-grained with one lock per runqueue. Only during balancing actions two runqueues get locked simultaneously. This is very similar to how Linux operates. Hence, a similar setup with a lock per SD should work as well. Idle and wakeup balancing already respect the system topology. Merely the statistics gathering and actual task migration have to be made aware of the hierarchical runqueues in Step 4. The periodic balancer, on the other hand, always balances the whole system. This becomes prohibitive with coscheduled sets, where each set would have to be balanced individually to balance the whole system. To keep the spirit of FreeBSD's balancing, this can be adapted as part of Step 5 so that always whole coscheduled sets are balanced (without drilling down into any child sets). FreeBSD's logic of running the global balancer on every nth clock tick on the first CPU, then translates to running the set balancer when the nth tick is charged to a set by its current master.

7.4.3 Comparison with Linux Integration

The presented FreeBSD integration should reach a similar level of functionality as the Linux integration presented earlier. Though, some questions regarding fine-tuning remain open at this point. For example, FreeBSD accounts CPU time in ticks. The collective context switch has to be integrated so that it does not cause a systematic skew in the accounting, which could happen if tasks are consistently preempted just before or after a tick elapsed.

Overall, it is difficult to say, whether the FreeBSD or the Linux integration is more complex. The integration focus is certainly different: The most difficult aspect of the Linux integration, adapting the distributed load balancer (step 4), is almost a non-issue in FreeBSD. Instead, FreeBSD requires more code to be written from scratch (task group mechanism, step 1) and the more difficult aspects are focused around the runqueue reorganization (step 3). In the end, mass might turn out to be the deciding factor: with roughly just 3,000 lines of code, the FreeBSD scheduler does not even reach a fifth of the size of the Linux scheduler – much less code to consider and to keep functional over the course of the integration.

7.5 Chapter Notes

The basic idea of Section 7.2 and a much more condensed description of a less developed version of the Linux implementation in Section 7.3 were previously published by myself in [84].

I decided to use FreeBSD as a second example, because of its decidedly different scheduler structure compared to Linux. While my initial decision for FreeBSD was based on a – as I later learned – outdated description of the scheduler, it still turned out to be a good choice. In particular, it helped to shape the minimal requirements for applying TACO, and it demonstrated once more that unbounded integers (the weight aggregation) can pose a problem, especially when your storage destination can only represent 64 distinct values.

Chapter 8

Analysis and Evaluation

In this chapter, the proposed coscheduler design is evaluated to see, whether it is able to kept true to the design rationales. The focus is hereby mostly on overhead that is introduced, as coscheduling – just by itself – has no inherent benefits. Benefits only come into play when exploiting one of the use cases, where advantages are gained due to coscheduling. Then, the benefits have to exceed the overhead to make the trade-off worthwhile. This chapter demonstrates this trade-off only exemplary for two scenarios, where coscheduling has been applied blind, i. e., without explicitly knowing whether exploiting coscheduling would yield any benefits. More intense evaluations of coscheduling benefits in general can be found elsewhere in the literature (see Chapter 2 for an overview and more specific pointers) and also in the third part of this thesis for use cases that were identified by the author.

To recap, there are four design rationales:

Versatility: Supporting only single use cases limits the potential and is not attractive for system designers.

Scalability: This is essential for today's and upcoming systems.

Interactivity: Many workloads today have an interactive component; coscheduling must not interfere with that.

Non-intrusiveness: Keeping scheduler behavior and most of the scheduler code intact, is the key for acceptance by system designers and application designers alike.

The most often expressed criticism concerns the lack of scalability. In research, this lack is proclaimed by supporters of implicit coscheduling schemes. In industry, this is indirectly visible in VMware's size limit on coscheduled sets. Finally, the last attempt to add something similar to coscheduling was shot down with that argument (cf. Section 1.2). However, the complete lack of coscheduling support – even in a small scale – in current general purpose operating systems silently points outs insufficiencies in the other areas.

The design presented in this thesis is specifically crafted around versatility and non-intrusiveness (cf. chapters 3, 6 and 7). While the other two rationales were also kept in mind, it is now time to take a closer look. In Section 8.1 the selected style of doing the collective context switch is analyzed. Afterwards, the issue of fragmentation in the hierarchical setup is considered in Section 8.2. Preempting larger coscheduled sets imposes additional costs, which are analyzed in Section 8.3. Finally, Section 8.4 demonstrates the Linux instantiation of the design in three scenarios. The first demonstrated use case is the scheduling of parallel VMs, where coscheduling avoids the very real problem of lock holder preemption. The second use case are parallel programs, where coscheduling potentially helps by enabling efficient fine-grained synchronization and an exclusive cache usage. Lastly, legacy capabilities are briefly checked by not using the existing coscheduling functionality.

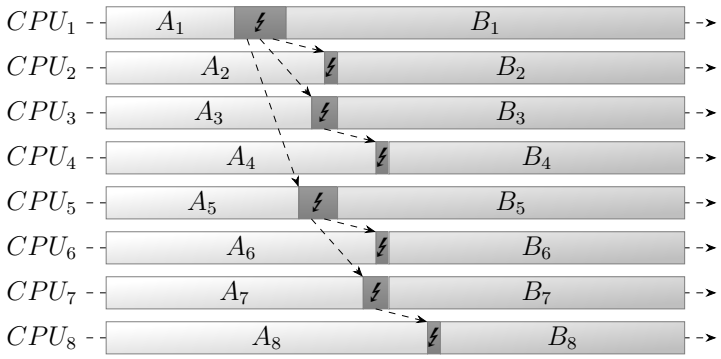
8.1 Collective Context Switch

Coscheduling in itself – especially the strict variant promoted by this thesis – makes the context switch more costly, because of the necessary synchronization. And compared to pure space partitioning, nearly every context switch can be considered overhead. This overhead ultimately determines the length of the used time slices: with context switch costs more or less constant, a longer time slice reduces the overhead relatively. However, longer time slices also put interactive usage at a disadvantage, so the length has to be selected carefully. While context switch costs have been examined in the past, multicore and especially coscheduling aspects have been ignored.

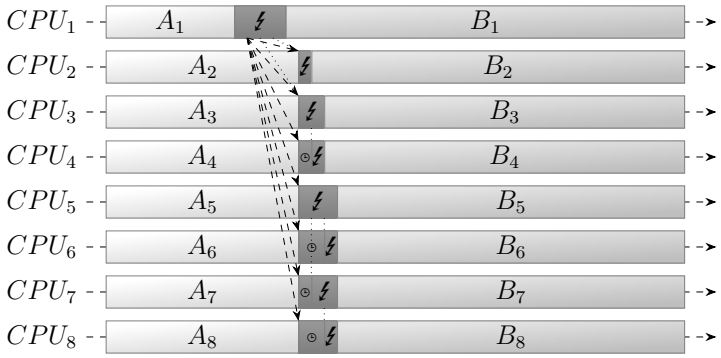
There are two types of costs associated with a context switch: direct costs caused by the execution of operating system code, and indirect costs caused by additional cache misses due to perturbed cache contents. Following Liu and Solihin [5], there are three categories of cache misses: natural cache misses, which would have occurred anyway (and thus, do not account towards context switch costs); cache misses, because some data was replaced while other tasks were running; and cache misses, because the original LRU sequence is reordered due to accesses of other tasks. Especially the latter becomes more problematic, when the task or tasks are specifically optimized towards the available cache size. While this cache miss analysis was done for uniprocessor systems, it can be transferred to shared caches as well – within reason.

8.1.1 Direct Costs

The direct context switch costs in the suggested coscheduling scheme are dominated by IPI latencies. Due to the tree-like notification scheme, it scales with the number of hierarchy levels, i. e., logarithmic in the number of CPUs. A context switch across a fictive octa-core system with a full binary tree as SD hierarchy



(a) Simple IPI dissemination along the SD hierarchy – a binary tree in this example. Minimal interruption per CPU, but synchronousness suffers and the scheme does not handle IPI processing delays well.



(b) IPI dissemination to all affected CPUs at once. Interrupted CPUs wait for the scheduling information to trickle along the SD hierarchy.

Figure 8.1: Direct context switch costs because of IPI latency.

is shown in Fig. 8.1a. Synthetic benchmarks on a quad-core Core i7 860 (SMT and turbo boost disabled) show that a collective context switch is usually finished within 2 μs, which corresponds to the IPI latency. So even without further optimizations, a five level hierarchy (at least 32 CPUs) would allow time slices of 1 ms while keeping the direct context switch costs below 1%. Such small time slices are inadvisable due to the indirect costs however. With larger time slices, say 10 ms, this scales to manycore systems with only negligible overhead. Nevertheless, the direct costs could be reduced with some optimizations. For instance, unnecessary hierarchy levels could be (temporarily) removed; or the notification mechanism could proactively send out IPIs to affected CPUs, whose first action is then to wait actively for scheduling information to arrive. The latter would take the IPI latency mostly out of the cost equation; it also handles variances in IPI delivery more gracefully. This is depicted in Fig. 8.1b.

Further optimizations are conceivable, like performing the scheduling decision on behalf of delayed CPUs on already available CPUs, but these would have to be carefully measured for their effectiveness.

8.1.2 Indirect Costs

The indirect costs caused by cache misses are harder to grasp. On the one hand, cache misses depend on the executed tasks; on the other hand, shared caches complicate that even further. If multiple tasks use a shared cache of a multicore processor competitively at the same time, they inevitably cause evictions of data of other tasks. Effectively, depending on individual memory access patterns and frequencies, every task sees a smaller cache. Technically, this is another source of natural cache misses, which affects all scheduling variants to some degree: time-sharing with and without coscheduling and even pure space sharing. Though, their occurrence varies with the scheduling scheme.

Consider coscheduling of independent tasks, for example in the context of resource contention avoidance: If a scheme is used, that creates many independent coscheduled sets, the indirect costs might actually increase compared to an uncoordinated scheduling: with every collective context switch a set of tasks is brought to execution, which probably will not find its data in the cache. Thus, all CPUs run into their bulk of cache misses at the same time and then they compete for memory bandwidth. With individual context switches on each CPU – either because the scheduling is uncoordinated, or the class based coscheduling variant is used – these cache refills are more spread out in time. With parallel applications with multiple tasks working on the same set of data, coscheduling generally improves the situation by reducing the number of applications that simultaneously access a shared cache. In the extreme case, coscheduling is used to assign caches exclusively to a parallel application during its time slice. This reduces natural cache misses due to space sharing to zero.

With some pessimistic assumptions, it is possible to create some (almost) worst case values for the indirect context switch costs based on cache size, cache/memory bandwidth and latency. In the worst case, all CPUs sharing a cache switch simultaneously and the whole cache has to be refilled. Either because time slices are relative large, or there were simply many other coscheduled sets scheduled in between so that no data is there to be reused. This cache refill can be done either with independent memory accesses, where the full memory bandwidth is the limiting component, or with dependent accesses (e. g., pointer jumping), where the achievable bandwidth is determined by the memory latency. This smaller bandwidth, however, can usually be realized per CPU, so that the actually realized bandwidth is again larger. Together with the cache size, this allows to calculate the time it would take to refill the cache. Every cache miss after a full refill would also have happened without a context switch.

These refill times were determined for two exemplary processors. The first is a hexa-core AMD Opteron 8435 (DDR2-533 RAM, 5 MiB L3 cache due to HT Assist). The achievable memory bandwidth is around 5 MiB/ms; with dependent accesses on all six cores, each core realizes an effective memory bandwidth of 0.7 MiB/ms. Thus, replacing the whole L3 cache takes about 1 ms for both access types. The second CPU is a quad-core Intel Core i7-2600 (DDR3-

1333 RAM, 8 MiB L3 cache, SMT), which has a higher memory bandwidth of about 15 MiB/ms, but is not much faster in the dependent case with around 0.8 MiB/ms for each of the eight CPUs. In this case, the dependent accesses form the slower case, but it also takes around 1 ms to refill the cache.

Note, that this refill delay is not realized en-bloc after being scheduled. More likely it is somehow spread out across the time slice. If the time slice is rather short or the working set rather small it might not even be realized fully. Additionally, not all of this delay actually constitutes indirect costs: some of the cache misses might belong into the natural category, especially if the application is not specifically optimized towards cache usage; and a scenario without context switches would also need time to process the data (e. g., fetching it from L3 cache instead of memory).

Within these uncertainties, it is possible to keep the indirect context switch costs on a multicore system below 1% with time slices around 50 ms to 100 ms if the system has similar characteristics as those above. Again, these indirect costs also apply to uncoordinated scheduling, which is only slightly better due to (probabilistically) better spread out memory accesses. Short experiments with the benchmarks, which are used later in this thesis, showed that most (but not all) also work with smaller time slices (e. g., 10 ms) without any measurable effects.

8.2 Fragmentation

With support for the different scheduling constraints, it is possible to realize all identified use cases – even simultaneously. It is just a matter of creating properly configured coscheduled sets and filling them with tasks. The coscheduler will handle everything thrown at it and give each coscheduled set the guarantees it requested. That said, simply creating coscheduled sets without considering system state at all might be a source of fragmentation, because of conceptually incompatible coscheduled sets.

Traditionally, fragmentation in parallel job scheduling refers to situations, where computational resources are unused despite the availability of tasks, because of shortcomings of the CPU allocation scheme. This is further categorized into internal and external fragmentation. Internal fragmentation occurs, when a partition is larger than required and the extra CPUs are unavailable for other allocations. External fragmentation refers to situations, where enough capacity is available to execute a certain job, but the scheduler is incapable to realize this. In both situations, CPUs stay idle unintentionally.

8.2.1 Internal Fragmentation

Internal fragmentation becomes a problem, when requests are arbitrarily sized and the allocation scheme cannot realize arbitrary partitions. When looking at the coscheduling use cases in Chapter 2, arbitrarily sized requests are uncommon.

Most use cases need to be aligned along the hardware topology, resulting in very few specific sizes. Only in some application design use cases it is possible to size applications independently from the hardware. And even there, it is not typical for an application to require exactly, say, 17 coscheduled CPUs. At these sizes, moldable or malleable applications can be expected, as most of today's and upcoming parallel programs are developed against some underlying parallel substrate. A fixed degree of parallelism is nowadays only expected in the lower single-digit range, e. g., small pipelines, or helper threads. A more relevant problem are applications with occasionally blocking tasks, essentially a variation of evolving applications. These do not always keep their partitions fully busy, and thus cause internal fragmentation – unless of course resource isolation is requested to prevent, e. g., cache interference.

Internal fragmentation is avoided by one of two techniques. The short term solution is to let idle CPUs within a coscheduled set pick up other work if allowed (cf. Section 6.4.1). The more efficient long term solution is to resize the coscheduled set and move it within the SD hierarchy (cf. Section 6.2). The latter is less cheap to set up, but is better with respect to external fragmentation.

8.2.2 External Fragmentation

Compared to other coschedulers on, e. g., clusters, external fragmentation is a considerable less pronounced problem, because coscheduled sets can be easily migrated within multicore systems. These migrations are also relatively cheap – except for migrations across NUMA domains, where the associated memory can be a problem. In the past, migrations were found to be the method of choice to handle external fragmentation, if it were not for the associated costs [72, 89]. Despite this, external fragmentation can occur. Either because the load balancer has not yet consolidated small SDs, or because the issued combination of coscheduled sets is conceptually incompatible due to unmatched partition shapes or sizes. An example for the latter are certain combinations of sets with the resource-sharing constraint and the resource non-sharing constraint; or independently selected degrees of parallelism of multiple parallel applications. Both may lead some holes in the schedule which cannot be solved by better packing.

It is the task of the operating system to avoid such unfavorable situations. It has to trade off partition sizes and shapes against the reasons for them being requested. Potential consequences of not matching a particular request were already discussed in Section 3.6. If these consequences are known and not considered too dire, partitions can be rearranged. When the definition of coscheduled sets is done by the operating system itself, all information regarding these trade-offs is available and fragmentation is usually not an issue, for example in setups that avoid resource contention. One way to handle this in the context of parallel application is presented later in Chapter 9. Also, with enough (varied) load in the system, fragmentation becomes less of an issue, as there will always be a small set or a task available, which can be executed.

8.3 Preemption

Besides realizing time sharing, preemption is the essential mean towards interactive behavior. Preemption allows to cut the time slice of the current task short in favor of another task that has become more important for some reason. This reason lies either within the preempting task and it could be something like a wakeup, migration, or priority change, or the reason could lie within the current task, which has become less important, e. g., because of a priority change.

As such, the concept of preemption transfers nicely into the SD hierarchy: instead of a task preempting another task, one SE now preempts another SE. The problem here is, that a scheduling entity usually represent a conglomeration of tasks and the tasks' properties must be propagated in some way within the hierarchy to be able to correctly assess the need for a preemption on a higher level. For a preemption check, these propagations need to be carried at most up to point in the SD hierarchy, where the paths towards the top of the changed task and the currently running task converge. If a currently running task encounters a possibly preemption causing property change, all runqueues towards the top, where the change needs to be propagated to, have to be checked for a now more important candidate.

With the proposed proportional share scheduling, the only property of a task that needs to be propagated upwards is its weight. And here, only the loss of weight of the currently executing SE might be a reason for a preemption, as it might suddenly be beyond its alloted time. However, with the CPU time tracking scheme suggested in Section 6.5, this check becomes moot. The other case, an increase in weight, cannot cause a preemption by itself. Hence, it is possible to postpone the weight propagation to reduce data structure contention on the higher hierarchy levels.

One particular issue is the coscheduling of interactive tasks with compute tasks, which for instance happens when unrelated tasks are represented by an aggregating SE in the next higher level. If not carefully coded, the interactive tasks may experience a drop in responsiveness, because the compute tasks consume the whole share of a group. This is the case in Linux (also without the coscheduling extension), because a preemption is only allowed, when a group has not used a substantial amount of its share, which the compute tasks have already done. (Counterproductive is here also the employed minimum length of a time slice, which – when combined with weight discrepancies – is able to prevent group execution for noticeable amounts of time.) In order not to fall victim to this, a group must always be allowed to consume its remaining share – without a lower limit on time slice length, which would give compute tasks the ability to use time beyond the remaining share. The CPU time tracking extension solves this nicely.

8.4 Experiments

To demonstrate the applicability of the proposed design in real life, the Linux implementation was subjected to different situations, to observe its behavior First, coscheduling of parallel virtual machines is examined in Section 8.4.1, which is probably the scenario where the effect of coscheduling is immediately noticeable. Then, parallel programs are coscheduled in Section 8.4.2. Finally in Section 8.4.3, the behavior of the coscheduling implementation is checked out, when nothing is explicitly coscheduled in order to assess the short-comings mentioned in Section 7.3.4.

8.4.1 Coscheduling of Virtual Machines

The first scenario considers parallel virtual machines. Specifically, multiple VMs do parallel compilations of Linux 3.0 kernels. Disk I/O was mostly avoided by using RAM disks within the guests and enough physical memory. All tests were conducted on an Intel Core i7 860 (quad-core, 2.8 GHz, SMT and turbo boost disabled) with 8 GiB RAM. Three different variants of scheduling VMs were evaluated in three different setups.

The three setups vary the number of CPUs per VM as well as the parallelism of the compilation within the VMs. Setup 1 has 4 VMs with 1,536 MiB RAM and 4 vCPUs each. The compilation is done at a degree of parallelism of 8 to ensure mostly full utilization. Setup 2 uses 8 VMs with 768 MiB RAM and 2 vCPUs each and a parallelism of four. Finally, setup 3 targets specifically partial load: 8 VMs with 768 MiB RAM and 4 vCPUs each are used, but the compilation is limited to at most two processes at once. In all cases, the task placement and load balancing within the VMs was left to the guest OS, i. e., there were no attempts to optimize something within the guest.

The evaluated scheduling approaches were Linux 3.2/KVM 0.14.1 without coscheduling, the same with coscheduling, and VMware ESXi 5.0.0-469512 as a more specialized solution. On the one hand, VMware ESXi is the only hypervisor with coscheduling capabilities. On the other hand, it is limited to this single use case due to being a hypervisor. Additionally, a baseline was measured for every approach and setup by executing a single VM with the approach in question. Based on theoretic bin packing of this baseline according to the setup, a relative performance index can be calculated, that shows the effectiveness of a particular scheduling approach.

The results are shown in Table 8.1. The deviation of individual experiments was significantly less than one percent after warm-up runs (except for the vanilla KVM runs). Thus, the results show the average of the runtimes of the compilation within all VMs and two runs after this warm-up phase. The first observation is, that the only non-coscheduling approach – vanilla KVM – is hopelessly behind in performance, which illustrates the extend of the problem that lock holder preemption poses for larger SMP VMs. Otherwise the results vary depending on whether the VMs are fully or partly loaded. For fully loaded

Table 8.1: Comparison of different VM scheduling approaches.

Setup	Approach	Absolute	Baseline	Relative
Setup 1	VMware ESXi	354.7 s	90.6 s	102.1%
(4 quad-core VMs,	cosched. KVM	382.6 s	94.7 s	99.0%
full load)	vanilla KVM	9,697.5 s	93.4 s	3.9%
Setup 2	VMware ESXi	716.8 s	176.3 s	98.4%
(8 dual-core VMs,	cosched. KVM	756.9 s	180.7 s	95.5%
full load)	vanilla KVM	1,208.8 s	180.6 s	59.8%
Setup 3	VMware ESXi	868.2 s	173.4 s	79.9%
(8 quad-core VMs,	cosched. KVM	1,484.0 s	175.7 s	47.4%
half load)	vanilla KVM	1,723.7 s	175.9 s	40.8%

VMs, the coscheduled KVM is only slightly behind the more specialized ESXi scheduler, after accounting for ESXi being faster in the baseline experiment. This difference can be attributed to the fact, that even the parallel compilation runs within a guest have some sequential phases. And ESXi is then able to execute multiple of these partially loaded VMs simultaneously – which also explains the relative performance of more than 100% in setup 1. The Linux coscheduling implementation used for this experiment did not yet support load balancing of small coscheduled sets, which made an automatic adaptation of the coscheduled sets to the actually used CPUs impossible and precludes better results. This limitation becomes significant in setup 3, where a partial load within the VMs is enforced, and the Linux implementation falls behind clearly. Though again, this is a problem of the implementation – not of the design.

8.4.2 Coscheduling of Parallel Programs

In this scenario, coscheduling was applied to several parallel applications selected from the OpenMP version of the NAS Parallel Benchmarks 3.3 [90]. The following benchmarks were selected:

EP.B: This benchmark is embarrassingly parallel. It neither includes significant synchronization nor does its dataset exceed the per-core caches. Thus, this benchmark serves as a reference case.

LU.W: This benchmark performs a LU decomposition on a rather small dataset. Contrary to all other NAS benchmarks, which use only the synchronization primitives of OpenMP for synchronization, this benchmark additionally makes use of active waiting.

CG.B: This benchmark is an implementation of the conjugate gradient method. It requires access to a larger dataset in short amounts of time, so that some contention within shared caches can be expected, when executed in parallel with other applications.

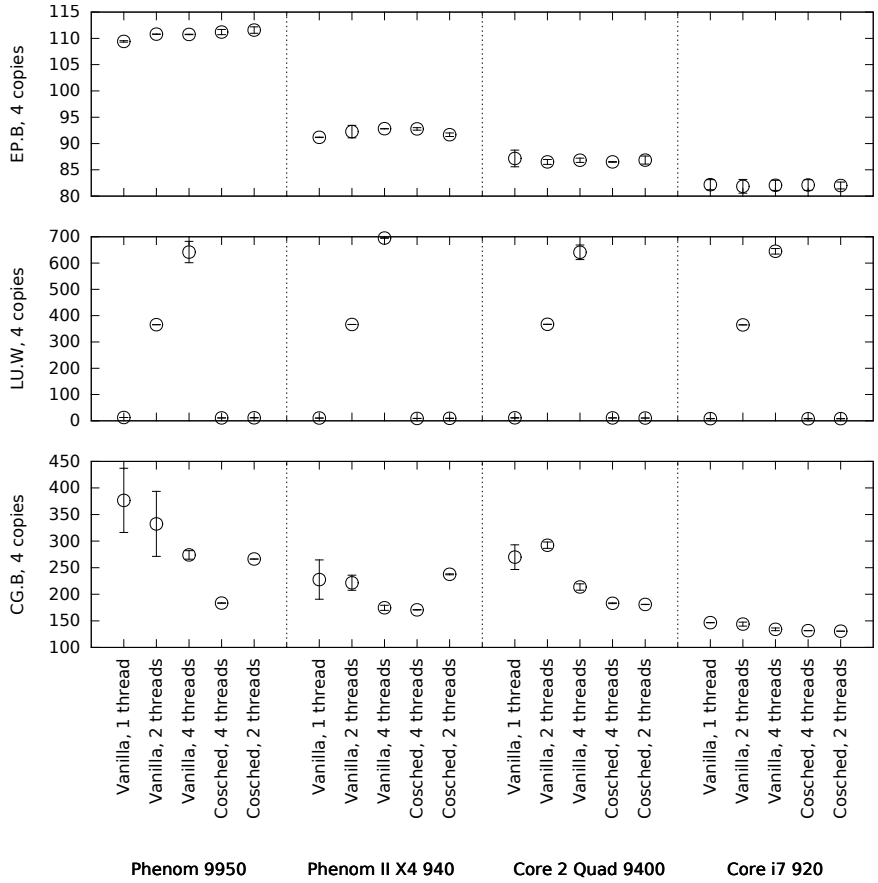


Figure 8.2: NAS parallel benchmarks, average completion times in seconds with standard deviation. For each benchmark, each point denotes the same amount of overall work – they only vary in the amount of threads and the employed scheduling strategy. While each experiment was carried out with active and passive waiting as another dimension, the diagram only shows the data point with the better performance, which is passive waiting for the multi-threaded vanilla cases and active waiting for the others.

For each benchmark, one to eight instances were executed in parallel using several different scheduling configurations and machines. The scheduling configurations include single-threaded execution of the benchmark instances as well as parallel execution with two and four threads per instance with and without having them coscheduled. Additionally, the OpenMP implementation was configured to use either active or passive waiting. Each experiment was run three times, measuring the time until all instances have finished. For this evaluation, four different systems were used representing different hardware generations of different vendors: an AMD Phenom 9950 (2.6 GHz), an AMD Phenom II X4 940 (3.0 GHz), an Intel Core 2 Quad 9400 (2.66 GHz), and an Intel Core i7 920 (2.66 GHz, SMT and turbo boost disabled). All system run Linux 3.2, either with or without integrated coscheduling.

An interesting subset of these results is shown in Figure 8.2. All points in a diagram represent the same amount of work, additionally each shown experiment is able to fully utilize a system. Thus, if it were not for overheads of certain configurations or architectural effects, there should be a horizontal line per diagram per system. The single-threaded execution has no parallel overhead. Thus, there is no difference between active and passive waiting. The multi-threaded, non-coscheduled runs with four benchmark instances cause the system to be overcommitted. In the diagram, the results for passive waiting are shown for these cases, which outperform those with active waiting. With coscheduling and the guarantee of synchronous execution of related threads, it is again possible to use active waiting, which avoids some overhead for synchronization heavy workloads.

EP.B being a compute heavy benchmark without substantial synchronization is an example for a parallel application, which is mostly oblivious to the employed scheduling strategy: no matter which scheduling configuration is chosen, the performance is always the same. This also shows that the coscheduling implementation does not cause significant overhead just by itself. LU.W uses active waiting at application level. That is, even with OpenMP configured for passive waiting, performance drops significantly as soon as the system is overcommitted and threads do not run synchronously anymore. Except for a non-parallel execution, only coscheduling is able to sustain the performance of such an application in an overcommitted system. CG.B represents an interesting third class of applications. With neither too much synchronization nor synchronization mechanisms outside of OpenMP, this benchmark can be efficiently executed with passive waiting as the results on the Core i7 demonstrate. However, these results are specific to the system architecture, specifically the cache architecture. On the other systems, the employed scheduling configuration makes a difference due to their respective effects on cache utilization. With multiple instances utilizing the same cache at the same time, performance drops. Without coscheduling, the number of simultaneous executed applications remains uncertain. Though, the more threads each instance has, the higher is the chance of accidentally coscheduling related threads, which are then able to share their data, reducing the cache pressure. The best performance for this benchmark is achieved by giving the last level cache in its entirety to an instance, i. e., to coschedule the instances across the whole processor. This also increases the predictability as these uncertainties are avoided. As a special case, the Core 2 quad has two last level caches: one for each pair of cores. Thus, it is sufficient to coschedule two thread instances to fully avoid contention within those caches.

Overall, the coscheduling implementation is able to realize its promised benefits. Though, it really depends on the application, whether the best performance can be reached by coscheduling only, or if other scheduling configurations reach similar performance levels. Admittedly, there probably are applications, where a

non-coscheduling configuration yields better results, but results with coscheduling will never be worse (except for overhead due to time-sharing) than executing those applications in isolation.

8.4.3 Unused Coscheduling Functionality

When no application makes actively use of coscheduling, the coscheduler logically degrades into the scheduler that it has been before. The runqueue hierarchy is mostly unused and application tasks are enqueued in the original runqueues. Still, there is some overhead associated with the different operations, a scheduler usually does. Some CPUs have to check the upper levels of the hierarchy to see whether new load has arrived and to do regular house-keeping on these runqueues. Enqueuing and dequeuing of tasks has to go up in the hierarchy until load on sibling runqueues is encountered. Due to a possible change in priority, this might even have to continue beyond this point to correctly assess preemption.

Most of this work can be avoided in case the coscheduling infrastructure is unused: upper levels contain at most one scheduling entity, which is always being executed. So, preemption is no issue and house keeping and checking upper levels can be postponed until additional load actually arrives. Especially the idea of postponing house-keeping already gains momentum in current operating system schedulers in form of tickless operation, which saves energy by allowing to keep CPUs longer in deeper sleep states. And in the special case of virtualized environments, it additionally reduces load in the host. This leaves actual enqueuing and dequeuing of tasks as main sources of overhead in legacy operation. Due to the hierarchical structure it could also be a source of contention, when multiple CPUs converge on the same runqueues. However, the hierarchy is unlikely to be traversed in a busy system as the bottom layer already contains load before an enqueue, or it still has load after a dequeue. The necessary reweighing is not necessary and can be done lazily. More critical is possibly a system that is semi-busy – with runqueues regularly transitioning between load and no load. Here, it is helpful when enqueuing and dequeuing have a fast-path that is realized without getting a lock, when there is nothing to be done besides reweighing.

In addition to this overhead, there might be slight behavioural changes in the load balancing due to load being balanced only between groups in the hierarchy. But this depends on how the original load balancer worked in the first place.

Looking at performance, several of the already described experiments were also run in non-oversubscribed scenarios, in which the coscheduling functionality – while still compiled into the kernel – remains unused. As the implemented coscheduler does not yet realize any of the described lazy or lock-free optimizations, these results give an idea of the overhead introduced by the runqueue hierarchy. For Section 8.4.1, this is equivalent to the baseline experiments, where only one VM was executed at a time. Comparing the kernel with coscheduling

to a vanilla kernel, we see a performance drop of 1.4% (94.7 s vs. 93.4 s) only for a fully loaded quad-core VM. The other two setups, which do not put the system into full load, do not show a noticeable difference (cf. Table 8.1). Part of the experiments from Section 8.4.2 were runs with just one benchmark instance getting executed utilizing four threads. Here, the kernel variant with coscheduling integrated was at most 1% slower than the one without coscheduling. Going further, the runqueue hierarchy code-paths were stressed with the messaging benchmark of perf (formerly known as hackbench) on the evaluation system from Section 8.4.1, showing a performance drop of 2.1% (11.19 s vs. 10.96 s for 10,000 loops).

8.5 Chapter Notes

The experiments from Section 8.4 have been previously published by myself in [84] and [91], though the analysis is slightly extended.

This concludes the second part of this thesis, where design and creation of the coscheduler itself was the main focus. In the upcoming final part the focus now shifts to the application of coscheduling. Individual chapters present new ways of utilizing coscheduling in multicore systems.

Part III

Advanced Coscheduling

where coscheduling boldly goes where no coscheduler has gone before

Table of Contents

Chapter 9

Management of Parallel Applications

The coscheduler design presented in the previous part of this thesis is principally able to handle all conceivable combinations of coscheduled sets and does a decent job of executing them. It is also able to use the leeway given by coscheduled sets, which do not specify the coscheduling constraint, to optimize the load distribution. So far, however, it was assumed that the specification of coscheduled sets is basically a one-way street: every use case just specifies coscheduled sets and the coscheduler just executes them. It completely ignores the fact that some applications might be able to change their behavior on demand. These behavioral changes can include, for instance, an adaptable degree of parallelism or interchangeable implementations of synchronization primitives.

In recent years, the need to go parallel in order to achieve high performance on multicore architectures [92] has stimulated not only the development of multithreaded software, but also the development of advanced parallel programming environments, such as *OpenMP* [2] or the *Intel Threading Building Blocks* [3], which ease future development of parallel applications significantly. Today's parallel applications are often moldable, so that they can fully utilize differently configured systems. Creating malleable applications is more difficult, but with sufficiently sophisticated parallel programming environments, which encapsulate enough domain-specific knowledge on parallel algorithms, the development of malleable or otherwise configurable applications is simplified quite a bit.

Thus, a scheduler from the near future will not only have to do as it is told, it will also have to steer applications towards a configuration, where the scheduler can reach its goal more easily. This chapter discusses, how TACO can be employed in such a scenario. First, Section 9.1 discusses the value of different application configurations and motivates the necessity of switching between them. Then, it continues in Section 9.2 with a description of configuration strategies and the resulting scheduling behavior. Afterwards, Section 9.3 evaluates TACO and compares it to several established approaches. The chapter closes in Sec-

tion 9.4 with a discussion of more specialized approaches and how they relate to TACO and general purpose multicore systems.

9.1 Management Considerations

There are several things to consider in order to determine a good configuration for a parallel application.

At first, there is the application itself. A parallel application is often characterized by its speedup: the ratio of sequential execution time to parallel execution time for a given number of CPUs. According to Amdahl's Law, speedup is at best linear, but usually it approaches an absolute limit due to non-parallelizable fractions within the code. Amdahl does not yet consider overhead that is introduced for the necessary management and coordination of more tasks. So, at some point there is not only no additional speedup but a slow down for each additional CPU. However, both lines of thought agree, that it is most efficient to execute a parallel application with only one CPU. Then, there is no communication or synchronization overhead and parts that are not that well parallelized have no negative impact on CPU utilization. And yet again, neither line of thought considers the effects of the system architecture on the speedup of applications, which may add a bit of zig-zag to the theoretically concave speedup function, as more CPUs may (or may not) result – for instance – in more cache and more memory bandwidth.

The point is, that each application has a certain ideal configuration, where it makes the most out of its allocated CPU time. This is often a single-threaded configuration, but it does not have to be. And every deviation from that configuration causes a loss in efficiency. Of course, a loss in efficiency does not necessarily mean a slower absolute execution time. Parallel computing exploits this very trade-off: the parallel overhead is more than offset by the parallel execution. With TACO's realization of fairness, the received CPU time of an application is kept constant, no matter how many CPUs are actually used. An application, that simply requests the whole system, will end up at a disadvantage compared to applications that operate at a more efficient point. For an application this means that it has to use the available CPU time as effectively as possible, which in turn means to revert to its ideal configuration – unless the operating system gives out CPU time for free for some reason.

There are three reasons for the OS to give an application extra CPU time:

- **The system is not fully loaded.** Unless the system is under severe restrictions with respect to power consumption, it is usually in the interest of the OS to finish jobs as early as possible. Hence, if CPUs are free, they should be put to good use.

- **There is fragmentation without deviation from ideal configurations.** This happens, when there is principally enough load to keep the

system busy, but the configurations of coscheduled sets prevent some CPUs from being used all the time. Reconfiguring some applications could fill these holes.

- **There is a load imbalance.** Some applications get more or less CPU time than intended and no balancing is able to change that. Reconfiguring an application might solve it. In order to give the application an incentive to change, additional CPU time can be offered. This trade-off can be worthwhile, when fragmentation can be avoided or less periodic rebalancing or periodic reconfigurations are needed in the future.

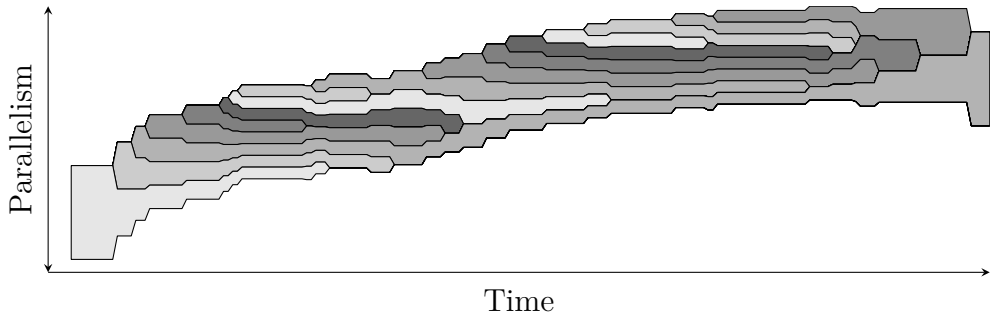
9.2 Management Strategy

Based on these observations, it is possible to apply a variation of the idea of equipartitioning, which is specifically adapted to TACO and allows to handle dynamic situations better than the traditional concept. Equipartitioning – on shared memory systems [93] as well as distributed memory systems [94] – tries to split the available computational resources evenly between running jobs. Time sharing is not used, which avoids overhead associated with context switches and makes it suitable for many use cases that require simultaneous execution. On the other hand, equipartitioning requires to reconfigure and remap running jobs whenever a new job arrives or a running job terminates.

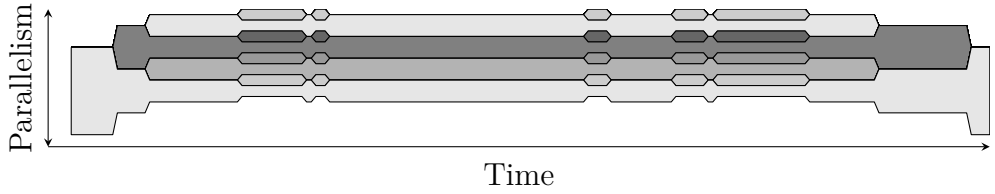
TACO supports the same (and more) use cases without requiring as many reconfigurations. Basically, TACO partitions CPU time instead of CPUs, so the actual size of a partition becomes a less deciding factor. Applications are generally executed in their most efficient configuration, unless doing so would be inefficient from a system point of view. Then, the malleability of applications is exploited.

9.2.1 Basic Equipartitioning with TACO

When there are not enough applications to keep the whole system occupied in their ideal configurations, the unused CPUs are distributed in form of additional CPU time among the applications. There exist many ideas on how much additional CPU time each application should get. For instance, one could give CPU time preferentially to those applications that use it most efficiently when executed with more CPUs. However, many algorithms require information about applications which might not be available at operating system level, or – if it is – they might not be able to handle real life speedups on multicore systems. Additionally, they all make the rather large assumption that the operating system is actually allowed to make decisions about the relative importance of different jobs. This assumption may be true in batch processing scenarios or with job dependencies. In current general purpose (and possibly multi-user) systems jobs tend to be interactive and independent from each other. And slowing down a job



(a) Reconfigurations with traditional equipartitioning. Each area corresponds to a running application. With every job arrival or termination all running applications are reconfigured.



(b) Reconfigurations with TACO. Each area corresponds to a partition in which applications are coscheduled. Applications are only reconfigured when the number of running applications crosses certain thresholds.

Figure 9.1: Schematic illustration of reconfigurations in the basic variants of equipartitioning and TACO. The width of the displayed band is constant; at every lateral movement of the band, all applications are reconfigured.

over-proportionally just because there exists a more scalable solution for another user's problem is unacceptable from a fairness perspective.

Thus, when applications are reconfigured to cover otherwise idle computation resources, the specified CPU time ratios are kept intact. Note, that this does not prevent an application from having an upper limit on how many CPUs it is able to use. Also, there can still be a component which restricts applications from reaching configurations below a certain efficiency when energy efficiency is considered. But such a component should be always active and independent from other running applications.

Traditional equipartitioning is prone to a lot of reconfigurations especially in dynamic scenarios. Whenever the number of jobs changes, every application has to be reconfigured, as illustrated in Figure 9.1a. Only at the cost of keeping parts of the system idle, an immediate reconfiguration on a job termination can be delayed. A variant of equipartitioning, called Folding (described in [94]), reduces the number of necessary simultaneous reconfigurations by always splitting the largest partition in half. Though, it still requires a reconfiguration on every job change and also creates larger imbalances in the CPU time distribution, which have to be addressed with periodic reconfigurations.

Listing 9.1: Reconfiguration logic for a basic equipartitioning scheme on top of TACO; balancing of applications is already handled by TACO.

```
int apps = 0;
int level = SYSTEM;

on_start(app a) {
        apps++;
        if (upper_threshold_reached(level, apps)) {
                level++;
                repartition_all_apps(level);
        } else {
                set_partition(a, level);
        }
}
on_terminate() {
        apps--;
        if (lower_threshold_reached(level, apps)) {
                level--;
                repartition_all_apps(level);
        }
}
```

With TACO, it is possible to go a step further and avoid most reconfigurations: by coscheduling multiple, identically shaped applications in the same place, a natural buffer against reconfigurations is created. New applications are simply coscheduled with existing applications without immediate reconfigurations, while terminating applications can often be removed without causing an immediate issue. Reconfigurations are only needed, when even load balancing cannot prevent a part of the system from going idle or when there are so many excess applications that it makes sense to generally use smaller partition sizes. Thus, reconfigurations are only necessary when the number of applications crosses certain thresholds. A schematic example is shown in Figure 9.1b. By adding a short-term hysteresis, i. e., having different thresholds for resizing up and down, it is possible to avoid frequent reconfigurations, when the number of applications fluctuates just around the threshold. This reconfiguration logic is expressed in pseudo code in Listing 9.1.

9.2.2 Optimizations

The reconfiguration logic does not have to be global as indicated in the previous section. It can also be applied locally per synchronization domain: when a single SD as acquired enough jobs, they are reconfigured to the size of the next lower level in the hierarchy; conversely, when a SD is idle (despite balancing) it prompts its siblings to return their jobs to their parent. This avoids system-wide simultaneous reconfigurations and enables an easier handling of individual ideal configurations or other individual constraints on coscheduled sets. Specifically, legacy applications (i. e., non-malleable applications) can be handled gracefully.

The ability to equipartition CPU-time while assigning different amounts of CPUs to applications, can also be used to reduce load imbalances further –

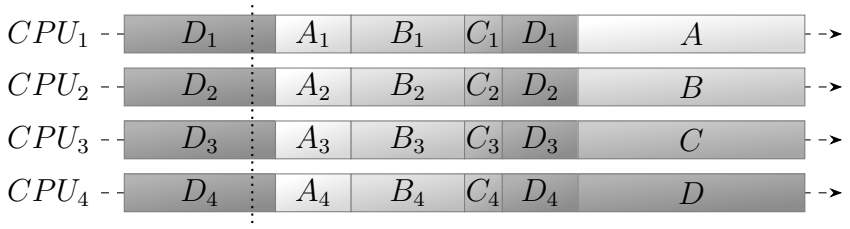


Figure 9.2: Switching to smaller partitions in TACO: If applications react within the next time slice, switching can happen without any transient oversubscription.

which – in turn – reduces the frequency of any rebalancing to ensure long-term fairness. For example, when there is a load imbalance between sibling runqueues, offending load can be dequeued from them restoring balance and enqueued in the parent runqueue instead. If applications do not scale linearly but do have a known speedup function, applications could receive a compensation in form of additional CPU-time for getting executed in a configuration other than their ideal one. However, this compensation should only get applied for deviations due to load balancing, not for freely distributed idle time. Otherwise, the result would be a "equi-speedup" scheduling scheme, in which the application with the worst speedup would receive most of the CPU time and use it inefficiently. For load balancing, if the compensation is considered too extreme, there is always the option to not resize that particular application and live with the imbalance instead.

Another benefit of using TACO as a building block for equipartitioning, is that reconfiguration delays of applications can be handled more gracefully than in traditional equipartitioning. Usually, as a side effect when applications are reconfigured during a job arrival or job terminations, there is a short moment of over- or undersubscription, respectively, until all applications have adapted to the new desired degree of parallelism. Depending on the scheduler, it may be possible to restrict applications immediately to the desired degree of parallelism, keeping the effects of oversubscription (which can be detrimental to applications requiring coscheduling) at least local to each application. TACO supports this, but it is also possible to go a step further: instead of enforcing new partitions immediately, applications are allowed a grace period during which coscheduling is used to keep reconfigured applications separate from not yet reconfigured applications. TACO will then react in the moment each application actually reconfigures itself. This way, any given coscheduling guarantees are kept and individual reconfigurations are decoupled from each other. For example, negative effects from resizing towards smaller partitions may be fully masked, when applications reconfigure themselves within one time slice: after each application within a SD runqueue has been notified of the intended change, execution of these applications continues as usual for a moment. As soon as an application has reduced its degree of parallelism sufficiently, so that it fits the next lower synchronization domain, it is preempted by TACO, draining the runqueue. If

all application manage the reconfiguration within one time slice, their execution will continue seemlessly as the only remaining SE in the runqueue will be the next lower level of synchronization domains. This is depicted in Figure 9.2.

9.3 Evaluation

In order to prove the applicability of equipartitioning with TACO, two variants of it were evaluated and compared to several standard approaches. Randomly generated workloads stress the malleability of tasks. Selected criteria for the effectiveness of an approach are the realized response time of a task compared to its isolated execution, the overall makespan, and the number of reconfigurations. The workload is described in detail in Section 9.3.1, followed by a description of all tested approaches in Section 9.3.2. The evaluation closes with the presentation and discussion of the results in Sections 9.3.3 and 9.3.4.

9.3.1 Workload and Evaluation System

The evaluated workloads are composed of several OpenMP applications taken from the NAS Parallel Benchmarks 3.3 described by Bailey et al. [95, 96] and developed by Jin et al. [90]. Only short running benchmarks (around one to three minutes when executed sequentially) were selected that are able to adapt the degree of parallelism at runtime, i. e., they repeatedly enter and exit parallel regions. Classifying these benchmarks according to their reconfiguration delay, there are fast adapting benchmarks (`bt.A`, `mg.B`, `sp.A`, and `ua.A`) and slow adapting benchmarks (`cg.B`, `ft.B`, and `is.C`). Benchmark `lu.A` is somewhat of a special case, as it is the only one that uses active waiting at application level. Table 9.1 gives more details about these benchmarks. All time related measurements in that table were obtained in absence of other interference. Thus, they are not valid when, e. g., memory bandwidth is shared with other applications, but they give a rough idea of the characteristics.

A workload consists of a selection of benchmarks with exponentially distributed inter-arrival times. That is, the jobs arrivals constitute a Poisson process. The benchmarks and their start times are randomly selected for each workload. The evaluation infrastructure then allows to replay a certain workload over and over again. Thus, it is possible to feed different scheduling approaches with identical workloads.

The evaluation system is a quad AMD Opteron 8435, a NUMA system with four six-core 45 nm K10 processors (codename Istanbul) clocked at 2.6 GHz. It has 64 GiB RAM (DDR2-533, 16 GiB per NUMA domain) and runs Linux 3.8 with NUMA memory balancing enabled. The used version of the GNU Compiler Collection – and thus also of GNU OpenMP – is 4.7.2.

Table 9.1: NAS benchmarks used for evaluation and their characteristics.

Benchmark	Description	Sequential exec. time	Speedup on partitions of size 2, 3, 6, 12, 24	Reconfigurations[1]	Average reconf. delay[2]
`bt.A`	Block Tridiagonal	94 s	1.8, 2.5, 3.8, 6.3, 13.4	1012	46 ms
`cg.B`	Conjugate Gradient	169 s	1.8, 2.5, 2.9, 5.5, 9.2	231	365 ms
`ft.B`	Fast Fourier Transform	82 s	1.7, 2.3, 3.1, 5.9, 11.7	112	365 ms
`is.C`	Integer Sort	52 s	1.9, 2.8, 4.7, 7.8, 12.2	16	1627 ms
`lu.A`	Low.-Up. sym. Gauss-Seidel	75 s	1.8, 2.4, 3.7, 6.2, 12.9	518	73 ms
`mg.B`	Multi Grid	13 s	1.2, 1.3, 1.1, 2.2, 4.3	1281	5 ms
`sp.A`	Scalar Pentadiagonal	71 s	1.5, 1.7, 1.8, 3.3, 7.0	3616	10 ms
`ua.A`	Unstructured Adaptive	68 s	1.6, 1.9, 2.7, 5.8, 15.9	36510	1 ms

[1]aka. parallel regions [2]when executed sequentially

9.3.2 Considered Approaches

Six different approaches were considered for this evaluation. The first two, *Uncontrolled Execution* and *Load-adaptive Execution*, are readily available on today's systems as they do not need additional support from the operating system: all decisions are made locally by the applications themselves. They represent the off-the-shelf baseline. *Standard Equipartitioning* and *Batch Processing*, on the other hand, are established approaches that require additional support. They form the conceptual baseline. Finally, there are *Topology-aware Equipartitioning* (TACO without coscheduling) and TACO itself.

Uncontrolled Execution (UE, UEp) This is probably the variant that is most often used today. Each application just considers itself, and the operating system is not aware of parallel or malleable applications. Thus, every application does what it wants and is not hindered by the operating system. In case of OpenMP applications, each application usually spawns as many worker threads as there are CPUs.

GNU OpenMP allows the user to select from three different waiting policies: passive waiting, spin-blocking (the default), and active waiting. For the experiments, spin-blocking (UE) and passive waiting (UEp) were used. The former results in applications that assume exclusive system access, while the latter sacrifices single application performance for overall throughput.

Load-adaptive Execution (LA, LAp) Another standard approach. The operating system is still not aware of parallel applications, but at least applications now recognize the fact that they do not own the system. Instead, they regularly poll the system load and adapt their own degree of parallelism. GNU OpenMP supports this style of execution when `OMP_DYNAMIC` is set. However, adaptations only happen when a parallel region is entered. Thus, it heavily depends on the program itself how often these adaptations take place.

In addition to that, achieved efficiency and fairness also depends on the load adjustment implementation and whether it uses additional sources of information. For instance, the load itself does not carry information about the number of concurrently running applications. Further, system load is typically adjusted only in terms of seconds; thus, it is not possible to react appropriately fast to thread creations and destructions. The load adaptation of GNU OpenMP is rather primitive, sizing the next parallel region to fill the free capacity according to the 15 minute load average. Load-adaptive execution was also tested in configurations with spin-blocking (LA) and passive waiting (LAp).

Equipartitioning (EQ, EQi) While not supported by current operating systems, a simple equipartitioning scheme was realized for this evaluation, which ignores for example the machine topology and other factors. That is, the available CPUs are simply divided by the number of applications and static assignments are performed until the next reconfiguration occurs. That said, care was taken to avoid migrations if possible, i. e., instead of blindly reassigning everything, CPUs are added to/removed from the set of CPUs already assigned to an application until the desired amount is reached.

The implementation of this approach uses one Linux CPU-set per application and explicitly manages their affinities; GNU OpenMP was slightly modified, so that it queries the CPU-set size for automatic reconfigurations. This approach is evaluated with the default NUMA memory policy (EQ) and with the memory interleave policy (EQi).

Batch processing (BP) While not useful in the considered interactive scenarios, batch processing gives another base line to compare the suggested approach to. Arriving jobs are simply processed in a FIFO order, one after the other. As the test applications do not have ideal speedups, this style of execution does not necessarily result in the shortest possible makespan.

Topology-aware Equipartitioning (TA, TAi) This is a variant of the suggested approach but without coscheduling. Compared to the simple equipartitioning scheme above, the partitions now respect the system topology, so that whole topological units or fractions thereof are used. This limits the possible partition sizes. For the quad-socket, 24-core evaluation system, this results in partition sizes of 1, 2, 3, 6 (one socket), 12 (half a system), and 24 CPUs (whole system).

Just like the simple equipartitioning scheme, this was realized with the help of Linux CPU-sets and a modified GNU OpenMP. Again, this approach is evaluated with the default NUMA memory policy (TA) and the memory interleave policy (TAi).

Topology-aware Coscheduling (TACO, TACOi) This is the suggested approach as described in Section 9.2.1. To gauge the principle applicability of

Table 9.2: Configuration of workload sets used for evaluation.

Set	Work-loads	Jobs per workload	Arrivals per minute	Application mix
A	8	40	6	all
B	8	40	9	all
C	8	40	9	all except `lu.A`
D	8	40	9	all except `ft.B` and `mg.B`

the approach, none of the ideas described in Section 9.2.2 were applied. Also, there is no hysteresis, i. e., the thresholds for switching partition sizes up and down are identical and correspond to the number of available partitions on a particular level. For the evaluation system and due to an implementation restriction, possible partition sizes are 1, 3, 6, 12, and 24 CPUs. This means if there is one application, it gets scheduled system wide, two to three applications are coscheduled on 12-CPU partitions, four to seven applications are coscheduled on sockets, eight to 23 applications use half-socket partitions, and finally 24 or more applications are executed as single-threaded programs.

This approach uses the Linux implementation of TACO as described in Section 7.3. Similar to the other approaches it is also evaluated with default NUMA memory policy (TACO) and with the memory interleave policy (TACOi).

9.3.3 Results

For the evaluation, different sets of workloads were analyzed against the different scheduling approaches. The workload sets differ in their application mix and in their average number of jobs. Their properties are given in Table 9.2. Each experiment was repeated five times, to see how stable the results are. For each experiment the makespan (i. e., the time the system is not idle while processing a workload), individual job slowdowns (i. e., response times normalized to isolated parallel execution times), and the number of reconfigurations were determined. These values are summarized in Table 9.3 with the best approaches highlighted.

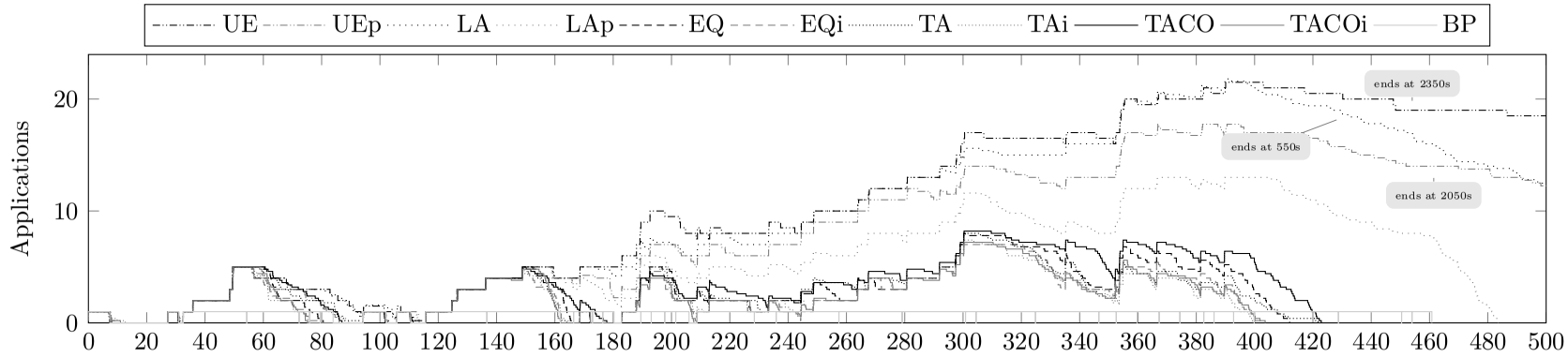
One exemplary workload of set A and one of set B is given in Figures 9.3a and 9.3b, respectively. They show the number of concurrently running applications over time. They are typical in that UE and LA generate unusually long makespans compared to the other approaches. This is due to the non-coscheduled oversubscription and applications making use of spin-blocking and, in case of `lu.A`, active waiting. LA is generally better than UE, as oversubscription subsides over time due to an increasing system load and a slowly reacting load adaptation. Switching the waiting policy of OpenMP to passive waiting, does not help significantly, as demonstrated by UEp and LAp, because of the active waiting at application level in `lu.A`. Only for workloads without any active

waiting, such as those in set C, UEp and LAp are almost competitive as shown in Figure 9.4a. Because of these results, there are no exhaustive experiments with UE/UEp, concentrating on the partitioning approaches instead.

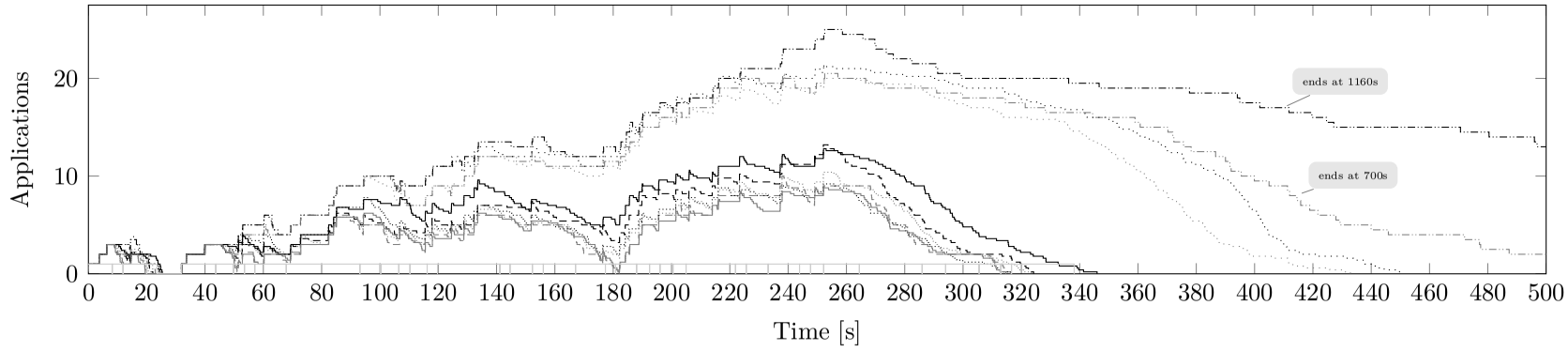
Considering EQ and TA, both using partitioning and no coscheduling, the results indicate that TA is better suited for workloads with a higher number of concurrent jobs than EQ. When there are not that many concurrent jobs, it is the other way around. This is because EQ ignores the topology of the system, and applications likely end up spread across multiple NUMA domains. With many concurrent jobs, EQ produces many remote memory accesses, while TA keeps accesses mostly within one NUMA domain. With only a few concurrent jobs, reconfigurations with TA are more prone to cause job migrations across NUMA domains separating the job from its memory. EQ does not have this problem this pronounced. This is supported by the results of their counterparts EQi and TAi with memory interleaving enabled: here, memory is distributed across NUMA domains in the first place and it should not matter where code is executed (except for multicore cache effects), making cross-NUMA migrations cheap at an overall increased cost for memory accesses. And indeed, EQi and TAi yield nearly identical results, outperforming their non-interleaving counterparts if (and only if) there are many cross-NUMA migrations, i. e., not many concurrent applications.

Comparing both topology-aware approaches with the default NUMA memory policy, it can be seen that TACO was not able to outperform TA – not even once. Another general trend is that EQ is also better than TACO. Though, this really depends on the actual application mix within a workload as illustrated by workload set D (see Figure 9.4b), where benchmarks `mg.B` and `ft.B` were removed from the application mix and the result favors TACO over EQ. Here, two effects accumulate: Benchmark `mg.B` is severely memory-bound and does not profit from multiple CPUs of one socket. That TACO issues larger partitions on average than EQ or TA is also not helpful. Instead, `mg.B` profits from the likely spread out execution of EQ. When paired with some other application that is not that memory-bound, `mg.B` has more memory bandwidth available. A similar argumentation holds for `ft.B`, though it scales a bit better.

Besides TACO issuing larger partitions, there is another difference to the other partitioning schemes: the TACO implementation relies on the Linux balancing mechanism to distribute load while TA and EQ do it themselves. Thus, the load balancing for TA and EQ is done centrally and proactively with minimal migrations per reconfiguration, while the load balancing for TACO is distributed and reactive in nature. This seems to cause more migrations than necessary, resulting in more remote memory accesses. This theory is supported by the results of TACOi, its counterpart with NUMA memory interleaving enabled. TACOi also issues larger partitions, yet it is nearly always better than TACO – especially considering that memory interleaving causes a performance degradation for TAi and EQi in workloads with a higher arrival rate.

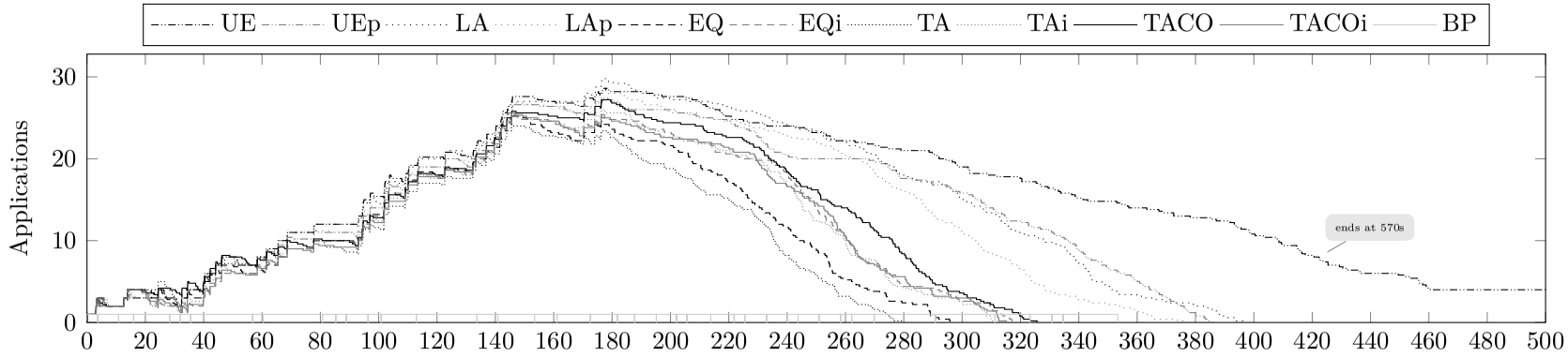


(a) Exemplary workload of set A. In this workload are five instances of `lu.A` with active waiting, which are responsible for the bad UE/UEp results. While the other approaches finish almost at the same time, the versions with memory interleaving realize more idle time during execution.

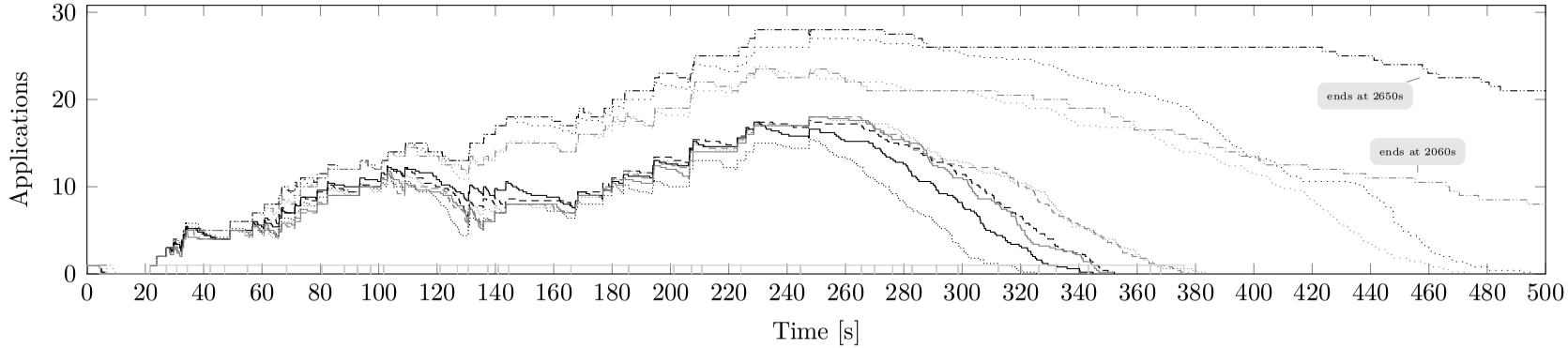


(b) Exemplary workload of set B. With only two instances of `lu.A`, UE/UEp finish earlier than in the exemplary workload of set A.

Figure 9.3: Selected workloads of the evaluated sets A and B.



(a) Exemplary workload of set C. This set has no active waiting at application level, but UE still suffers from OpenMP's spin-blocking.



(b) Exemplary workload of set D. In this set, some workloads are particular bad for EQi and TAi. In other workloads, the performance of EQi and TAi is comparable to EQ or TACOi.

Figure 9.4: Selected workloads of the evaluated sets C and D.

In fact, TACOi is the best of all evaluated approaches for set A, and a close second after TA for set B. In set C, benchmark `lu.A` is missing, which is very sensitive with respect to interferences on the L3 cache of the evaluation system and profits from having cache just for itself. Without this advantage, TACOi comes in third after TA and EQ. In set D, TA is first followed by TACO and TACOi.

9.3.4 Exploring the Design Space

In addition to the presented results, some additional experiments were conducted to explore the design space of the topology-aware scheduling schemes. Foremost, for the topology-aware approaches being competitive on NUMA systems, a mechanism is necessary that keeps memory and tasks close together. If such a mechanism does not exist, topology-aware schemes often separate tasks from their memory and an approach like EQ or enforced NUMA memory interleaving are actually better, as there is probably at least some memory allocated at the NUMA node(s) where the tasks are executed. With Linux 3.8 a very simple NUMA memory balancing mechanism is available, which periodically enforces a migrate-on-next-touch policy to move memory to where it is needed. While not perfect, it helps not only TA and TACO, but also EQ and delivers consistently better performance in the conducted experiments than doing nothing.

With TACO/TACOi there is the additional freedom to restrict allowed partition sizes without running into fragmentation or fairness issues. The presented results use a mostly unrestricted set, where partitions of sizes 1, 3, 6, 12, and 24 CPUs are allowed (due to an implementation restriction, supporting 2 and 3 at the same time is not possible). Running experiments with more restrictions, allowing only partition sizes of 1, 6, and 24 CPUs – core, socket, system – did not give good overall results: only `lu.A` really profited from this and the basic implementation does not yet allow for individual exceptions. For a similar reason, no advanced balancing logic as proposed in Section 9.2.2 was realized, as it requires application knowledge to be effective. Instead, periodic rebalancing was used for TACO and TACOi, as this is what the Linux scheduler uses to resolve imbalances and into which the coscheduling support is tightly integrated. While this worked fine for TACOi, the default load balancing settings had to be slightly modified for TACO, so that rebalancing applications across NUMA domains is kept at a very low frequency. Otherwise the resulting remote memory accesses and triggered page migrations can quickly kill the performance.

In Section 8.1 the context switch costs and there influence on the optimal time slice length were discussed. Linux itself does not used fixed time slices, but adjusts them according to task weight, current load and the number of CPUs in the system. For the evaluation system, this translates to time slices from as short as 3 ms up to 24 ms depending on circumstances. Paired with a regular 100 Hz timer, the experiments with TACO and TACOi should result in time slices of 10 ms to 20 ms most of the time. Experiments with specific setups show,

Table 9.3: Averaged results of different approaches per set.

Set	Approach	Makespan (BP=100%)	Average Slowdown	Reconfig-urations
A	BP	100%	5.7	0
	LA	134%	20.3	n/a
	LAp	120%	13.0	n/a
	EQ	87%	4.1	145
	EQi	**84%**	**3.8**	**132**
	TA	94%	4.3	76
	TAi	**85%**	**3.8**	**69**
	TACO	99%	5.5	70
	TACOi	**83%**	**3.7**	**59**
B	BP	100%	10.2	0
	LA	134%	22.8	n/a
	LAp	127%	19.0	n/a
	EQ	94%	8.4	170
	EQi	94%	8.2	167
	TA	**91%**	**7.3**	**103**
	TAi	95%	8.3	98
	TACO	100%	10.4	64
	TACOi	**91%**	**7.7**	**63**
C	BP	100%	10.8	0
	UE	198%	31.5	0
	UEp	113%	15.4	0
	LA	120%	18.4	n/a
	LAp	110%	14.4	n/a
	EQ	91%	8.5	173
	EQi	94%	9.0	172
	TA	**89%**	**7.7**	**106**
	TAi	93%	9.0	98
	TACO	98%	11.7	68
	TACOi	92%	9.0	75
D	BP	100%	11.9	0
	LA	132%	29.7	n/a
	LAp	127%	24.5	n/a
	EQ	92%	11.1	167
	EQi	102%	13.0	167
	TA	**86%**	**8.6**	**112**
	TAi	100%	12.7	106
	TACO	89%	10.3	51
	TACOi	91%	10.6	59

that some of the NAS benchmarks gain a few percent more performance with an increased average time slice length of 50 ms to 100 ms. However, this relies on specific partition sizes and specific coscheduled applications. For the more randomized workloads in this evaluation, it did not translate into a measurable advantage.

9.4 Related Work

From the vantage point of the system, partitioning causes less overhead than coscheduling: there are less context switches and there is no need to achieve a simultaneous context switch across multiple CPUs which usually does not scale.

Coscheduling, on the other hand, has no reconfiguration overhead making job arrivals and terminations very cheap. Instead, there is overhead from time-sharing and reduced efficiency due to normally sub-linear speedups.

The idea of operating system enforced fairness between multiple parallel applications is not new. A pioneering work is [93], which introduces *Process Control*: a method to fairly distribute the available CPUs among running parallel applications. It includes a concept of malleability and also considers non-malleable applications by reducing the pool of available CPUs for malleable applications accordingly. CPUs are distributed in a round robin fashion, until either an application reaches its individual maximum or no more CPUs are left. The approach does not consider the system topology in any way, but for the targeted early shared memory systems this does not really matter.

On distributed memory systems, on the other hand, topology has always been important. In [94], two concepts for such systems are presented: *Equipartition* and *Folding*. Equipartition conceptually splits a regular, non-hierarchical system topology (e. g., a grid) into connected, almost equally sized partitions. Folding always splits the largest partition in two halves (with, e. g., hypercubes in mind). This has the benefit of avoiding parallel reconfigurations. The more unfair distribution of CPU time is countered with periodic rotations of applications. Folding is also recognized as a possibility to make rigid or moldable applications pseudo-malleable: due to the halving of partitions, non-malleable applications experience always a doubling of threads per processor, which works reasonably well as long as there is not much synchronization. Both approaches do not consider any form of coscheduling. However, as far as partition sizes are concerned, the suggested TACO-based approach is quite similar to Folding. For example, the idea of pseudo-malleable applications can be transferred without problems. Contrary to Folding, TACO is able to achieve a fair CPU time distribution without periodic rotations by coscheduling applications of different sizes.

Corbalan et al. suggest *Compress&Join* [97], a combination of coscheduling and partitioning, where job malleability is used to reduce fragmentation normally associated with coscheduling: based on an ideal number of processors for each

application, their approach fits multiple applications into a coscheduled time slot, possibly sizing them down a bit with a bounded deviation from the ideal size. Fairness and system topology are not considered; and while exclusive resource usage due to coscheduling is mentioned, it is not considered when partitioning a time slice. Bhadauria and McKee [98], on the other hand, consider fairness and resource contention in their partitioning scheme. Similar to Corbalan et al., they also use partitioning within coscheduling. However, they use a sampling and feedback mechanism to intelligently select and size applications to be scheduled simultaneously, so that contention of system resources is hopefully minimized. A hierarchical system topology is not considered. Both approaches require large time slices (measured in seconds) and long running applications. Contrary to that, TACO works with short time slices (measured in milliseconds, similar to usual OS time slices) and does not disturb interactive behavior. TACO's nesting of time and space slicing only requires partition wide synchronization (instead of system wide synchronization) and enables variable length time slices. Additionally, it recognizes hierarchically arranged resources. While the evaluated TACO-based equipartitioning scheme did not consider application speedups and did not arrange for specific applications to run simultaneously, TACO itself is flexible enough that these features can be easily added.

9.5 Chapter Notes

Parts of this chapter have been published previously by myself in [99] and [100] – with the latter being an extended version of the former. In particular, the evaluation presented in Section 9.3 and the related work from Section 9.4 have been taken verbatim (apart from small changes to allow for a better text flow) from [100].

Chapter 10

Scheduling with Turbo Boost

After processors stopped getting faster, they started to get parallel. Software, on the other hand, is still catching up. One reason is certainly the large amount of single-threaded legacy software; another reason is that development of parallel programs is still hard. This has lead to the weird situation, that existing software got slower when executed on newer hardware, because the available energy budget was split statically and evenly between cores of a multicore processor – which in turn meant a decrease in frequency. Thus, if you wanted to buy a computer system, you had to trade off parallelism against single-thread performance at that point. The hardware manufacturers reacted with processors, where the energy budget is distributed dynamically between cores instead of statically. Hence, if one or more cores have nothing (or not as much) to do, the available energy headroom is reduced by increasing the frequency/voltage of the processor – with all decisions done in hardware. Roughly speaking, energy consumption becomes a constant while frequency and voltage are now variables. Depending on the manufacturer, this technique is called, for example, *Intel Turbo Boost Technology* [101] or *AMD Turbo Core Technology*. This thesis sticks to *turbo boost* as it is the prevalent term for this kind of technique.

Turbo boost essentially turns energy into a resource shared among cores of the same processor. The parallelism/single-thread performance trade-off became (to some degree) a run-time decision: the less you execute in parallel, the faster it is. Of course, if you have multiple things to execute, usually you will still be finished earlier with all of them, when you execute them in parallel, because the frequency increase does not beat the power of parallel processing. However, not always are all tasks equally important – a situation in which current operating systems are not able to exploit the full potential of turbo boost. They take longer to execute the important part of a workload while consuming more energy than necessary. This chapter applies coscheduling to give operating systems the ability to take full advantage of turbo boost. Section 10.1 starts by describing the issues in more detail. Section 10.2 outlines the solution with coscheduling, which is then is evaluated in Section 10.3. The chapter closes with a discussion of related work in Section 10.4.

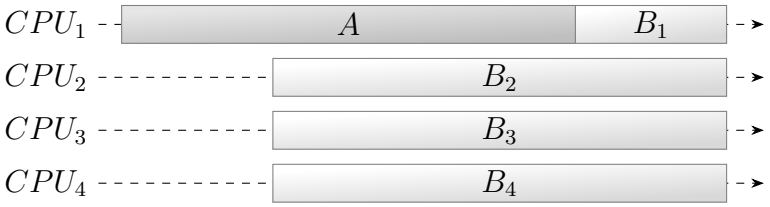
10.1 Issues with Turbo Boost

The major advantage of turbo boost, the dynamic boosting of processor frequency as long as the processor operates within certain physical limits, is also its downside. The increase in single-thread performance comes at the cost of a superlinear increase in power consumption. While that is a trade-off one is often willing to make, turbo boost in its current realization is responsible for two problems in computer systems that support it. The first problem stems from the fact that turbo boost is realized in hardware and is transparent for the operating system. This confuses the accounting within the scheduler (or is outright ignored) and leads to suboptimal scheduling decisions. The second problem is shared with its software-driven variant dynamic voltage/frequency scaling (DVFS): there are not enough hardware knobs to apply turbo boost selectively to only those tasks that actually profit from it – making it less effective.

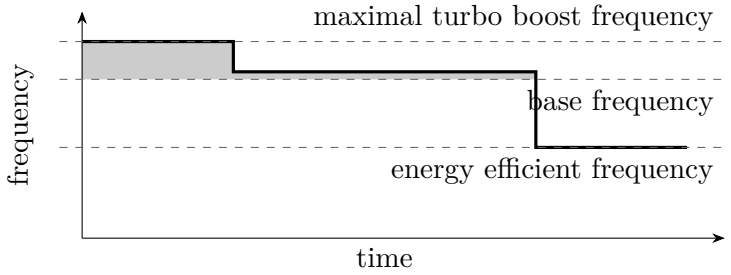
To give concrete examples, consider the evaluation system used later in this chapter. It has an Intel Core i7 860 processor, a quad-core processor which has a base frequency of 2.8 GHz and a 1/1/4/5 turbo configuration. That means, you are guaranteed to be able to utilize all four cores at the same time with them sustaining a frequency of at least 2.8 GHz. However, if there is thermal/energy headroom, the processor may increase the frequency by up to 1, 1, 4, or 5 steps of 133 MHz given that not more than four, three, two, or one core are active simultaneously, respectively. Thus, with a single core active, this processor can run at a frequency of 3.47 GHz. With all cores active, the upper limit is 2.93 GHz. The specific system used for evaluation was usually able to execute its workload at the maximum frequency, except in cases with all four cores being active where the frequency oscillated between 2.93 GHz and the base frequency of 2.8 GHz.

Considering workloads of tasks of mixed importance and an operating system scheduler without any specialized support for turbo boost, this leads to a worst case where an important task is slowed down by 15% – just because the scheduler decided to execute some background jobs in parallel on otherwise unoccupied cores, which limited the achievable frequency. Even using tools such as `nice` in Linux will not help, as typical SMP schedulers cannot handle this kind of interdependency and leave computing capacity idle on purpose. Continuing this example, in addition to the slowdown of the important task, it means that the background jobs will be executed while turbo boost is active – with the highest possible energy consumption. This situation is illustrated in Fig. 10.1. Ideally, background jobs would be executed in an energy efficient manner, but switching the processor to an energy efficient configuration would slow down the important task even more, as there is just one frequency/voltage domain in Intel processors. (AMD allows a bit more freedom with frequencies, but has the same restriction for voltage in most processors, which is the more important of the two in terms of energy consumption.)

Even if there is a phase in the workload with only background jobs active, typical operating systems have to execute them with activated turbo boost, if



(a) Schedule of an important task A and background load B.



(b) Frequency transitions for the given schedule.

Figure 10.1: Schematic illustration of problems caused by turbo boost on an Intel Core i7 860. At the beginning, the important task A is able to run at the highest possible frequency. When background load B appears, the performance of A drops while more energy than necessary is spent on B. Only when A vanishes, it is possible to switch the processor to an energy efficient configuration without hurting A even more.

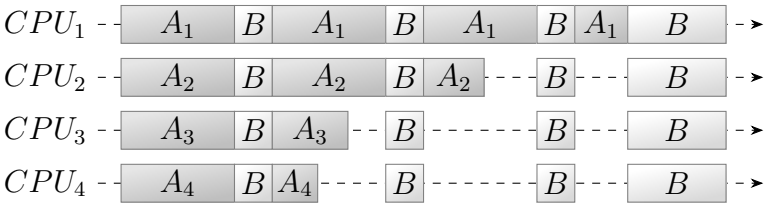
a noticeably slow down of interactive tasks is to be avoided. This is because frequency/voltage transitions are usually still triggered reactively, for example with the `ondemand` governor in Linux or its Windows equivalent [102, 103]. Some measurements to that effect and possible solutions are presented in [104, 105]. However, since the introduction of turbo boost at the latest, processors adapt new frequency/voltage settings fast enough to make transitions on context switch feasible. This is exploited by the approach presented in the next section.

10.2 Turbo Boost with Coscheduling

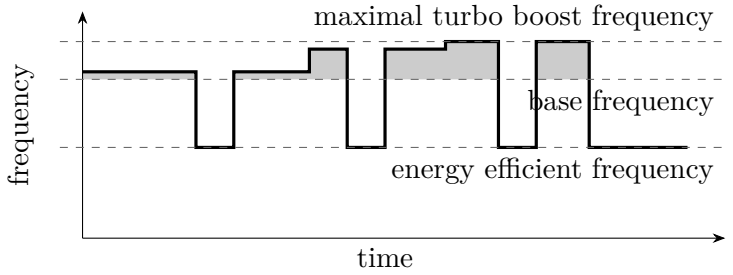
There are two problems to address to make turbo boost suitable for all kinds of workloads:

1. Ensure that certain classes of tasks are executed without wasting energy. That is, disable turbo boost selectively and switch to an energy efficient frequency/voltage setting for these tasks.

2. Prevent certain classes of tasks from stealing from the energy budget of other classes of tasks. That is, do not slow down certain tasks excessively.



(a) Schedule of important load A and background load B.

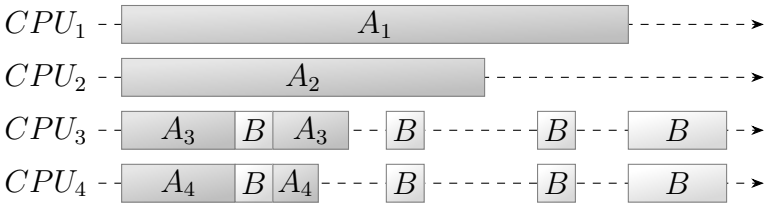


(b) Frequency transitions for the given schedule.

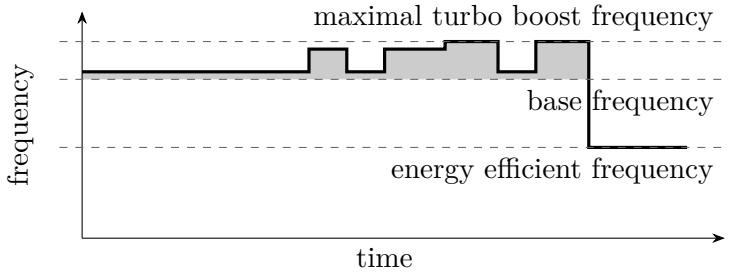
Figure 10.2: Coscheduling of background load on an Intel Core i7 860. Coscheduling enables to lower the processor frequency and execute the background load energy efficiently. In phases of low parallelism, the important load can take full advantage of turbo boost.

The first is achieved by coscheduling those classes, and a small scheduler extension to support frequency transitions on context switches. More specifically, each class is represented by a coscheduled set with a close placement and a minimal concurrency constraint, so that tasks – if there are many – appear in groups, with each group being able to fully occupy a processor. This becomes important for multi-processor systems, as each processor (or socket) has its own energy budget and is its own isle for frequency/voltage scaling. Applied to the example workload of important tasks with background load, this turns an uncoordinated schedule, where it is unlikely that enough tasks line up at any time to lower the frequency, into a coordinated one as depicted in Fig.10.2 – with designated phases where the frequency can be lowered. Note, that different classes may have different target frequencies for an energy efficient execution. By using multiple coscheduled sets, this scenario is easily supported.

If there are not enough tasks in a class to fully occupy a whole processor, the remaining tasks will still be coscheduled within the processor. But then the coscheduled set may get executed alongside other tasks, which may desire different target processor frequencies. In that case, the highest frequency setting among the simultaneously executed tasks will prevail: on current processors frequencies/voltages are configured per CPU via ACPI P-states, and processors typically chose the P-state, which provides the highest performance among those configured on the active CPUs. While this does prevent reaching a lower fre-



(a) Schedule of important load A and background load B.



(b) Frequency transitions for the given schedule.

Figure 10.3: Coscheduling of light background load on an Intel Core i7 860. The important load is prioritized in the frequency selection. Additionally, the background load is only allowed its share of CPU time despite available capacity.

quency in case important tasks are executed at the same time, it is still better in terms of energy and execution time than forcing non-simultaneous execution with an additional isolation constraint, where parts of the system would stay idle and some tasks would not get executed. Due to the dynamic weights described in Section 6.5, the background load will not be allowed to execute just because idle computing capacity is available. Instead, the background load will only be able to use its assigned share. This prevents the background load from permanently using up a portion of the energy budget, which is needed by the important tasks to reach peak performance. An example for this is given in Fig.10.3.

The coscheduler also presents some other configuration options to realize similar but not quite identical settings. While they might be beneficial in specific scenarios, they lack the generality of the presented configuration. For example, instead of having the background load contained in a coscheduled set, one could create a coscheduled set for the important tasks. If both groups have many tasks, it would just look like the left hand side of Fig.10.2. While for a low number of important tasks the idle selection described in Section 6.4 would kick in, so that for any amount of background tasks it would look like the right hand side of Fig. 10.3. However, it would not be efficient for many important tasks paired with few background tasks. Also, it makes the selection of what the energy efficient frequency should be more complicated – the frequency selection is briefly discussed as part of Section 10.4. A similar argument holds for an

alternative solution without coscheduling, which relies only on per task target frequencies activated on context switch and enforcing system wide weighted fairness as described in Section 6.5, which would dequeue the background load from time to time. This scenario would look similar to Fig. 10.3 but without synchronized execution of background tasks. Thus, opportunities to lower the frequency in the presence of foreground load are missed, and it would not be possible to have different classes of energy efficient tasks.

10.3 Evaluation

The presented approach was evaluated with a simple scenario, consisting of a foreground load, which the user is actively waiting for to finish, and a background load resembling other activities of the operating system, where speed does not matter that much and which is not the main reason for leaving the computer system powered on. The background load has already been declared as unimportant by means of `nice -19`, which gives the background load about 1.5% of CPU time compared to the foreground load, when executing both on the same CPU. This evaluation now studies the effects of segregating both loads via coscheduling for different degrees of parallelism of foreground and background load.

The foreground load consists of compiling an `allmodconfig` Linux 2.6.31.4, while the background load consists of repeatedly computing some MD5 checksums across large files. The degree of parallelism is adjusted by `make -j`*N* for the former and by just forking an appropriate amount of workers for the latter. Both loads are executed within RAM disks to avoid the influence of disk I/O for this evaluation. The evaluation system is an Intel Core i7 860 (2.8 GHz base frequency with a 1/1/4/5 turbo boost configuration) with disabled SMT and 8 GiB RAM. Depending on the type of test run, the system either executes on top of vanilla Linux 3.0 or Linux 3.0 with an early realization of the coscheduler described in Chapter 7. The coscheduler was extended to store a target frequency per coscheduled set, which is set on a context switch to that particular coscheduled set. Energy measurements were automated with a Gude Expert PDU Energy 8001.

Due to some short-comings of the used coscheduler realization, the setup differs slightly from the one presented in Section 10.2. Both, foreground and background load, are placed in their own coscheduled set. Additionally, each coscheduled set has an isolation constraint. While this fully prevents simultaneous execution of foreground and background load as illustrated in Fig. 10.3, it forces the background load into its share of about 1.5% CPU time. (The used coscheduler does not support dynamic weights.) The coscheduled set of the foreground load is configured to use turbo boost, while the background load uses a frequency that was determined a priori to be the most energy efficient for the particular background load. This was done by determining the power consump-

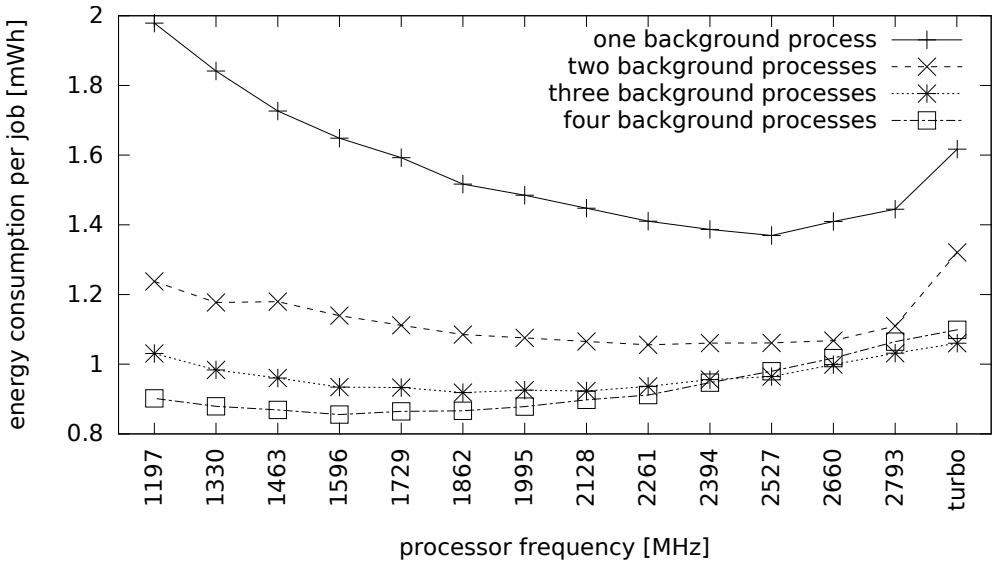


Figure 10.4: Energy consumption for one unit of background load for different degrees of parallelism and different frequencies. The idle power consumption of 52.2 W has been factored out as fixed costs.

tion of the system when idle (52.2 W), and then determining the additionally needed power when executing the background load with various degrees of parallelism for each possible frequency setting. From this, it is possible to derive the energy consumed by an individual background job (excluding idle consumption as the system is considered to be powered on with or without background jobs). These results are shown in Fig. 10.4. Ways to determine target frequencies more automatically are discussed briefly in Section 10.4.

Experiments were done for degrees of parallelism one to four for both, foreground and background load, i. e., 16 combinations. To compare the coscheduled variant with the non-coscheduled variant, the reference case with a vanilla Linux is measured first: For a certain combination of foreground and background load, both are started in parallel. The moment the foreground load finishes, the required time, consumed energy, and the amount of finished background jobs is recorded. The latter is needed to construct an equivalent execution with coscheduling: in the coscheduled version the background load will make less progress in the time needed by the foreground load. In order to keep the energy measurement comparable, the execution of the background load must continue until the same amount of work as in the reference case has been finished. However, as the execution time of the background load is of no importance, time measurement is stopped when the foreground load finishes – just as in the reference case.

The gains and losses of performance and energy consumption of the coscheduling variant compared to the reference case are shown in Fig. 10.5. Additionally, the energy delay product is shown as an indicator whether the perfor-

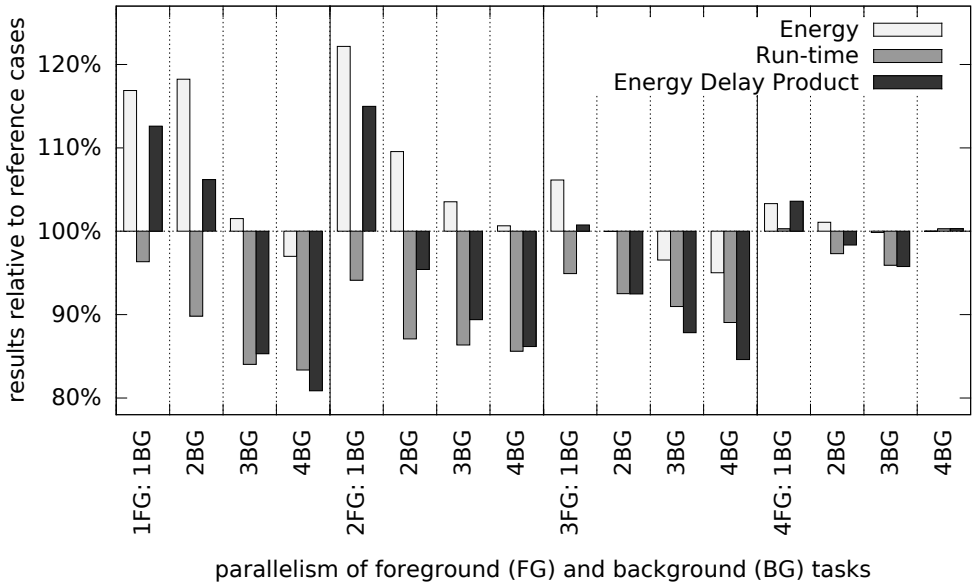


Figure 10.5: Turbo boost optimized scheduling compared to turbo boost agnostic scheduling. Using coscheduling to optimize for turbo boost is always beneficial for performance, while from an energy perspective it only makes sense with a parallel background load.

mance/energy trade-off is worthwhile. The isolation of the foreground load from the background load results always in the expected win in performance – except when foreground load can fill the whole processor on its own; then it makes no difference. For energy, this is more complicated: On the one hand, the separation makes the foreground load not only faster but also increases the energy consumption disproportionally, i. e., overall the computation of the foreground load needs more energy than before. On the other hand, the computation of the background load is now done at an energy efficient frequency, which counters the previous effect. However, only for background load spanning more or less the whole processor this can offset the balance enough, so that there is an overall reduction in energy consumption. For all other cases, the gain is not enough.

Considering the trade-off between performance and energy as expressed by the energy delay product, there are some more acceptable cases where – while not actually saving energy – the additional energy costs go along with a sufficiently increased performance. The rules of thumb, that can be derived from this evaluation are:

1. If your goal is performance, you should always apply coscheduling to give your important load that extra bit of oomph on turbo boost enabled systems.

2. If you accept only a moderate increase in energy consumption for more performance, you should abstain from splitting your load, when the load – even before splitting – is unable to fully utilize a processor.

3. Finally, for energy optimized systems, background load should only be coscheduled, when it can fully utilize a processor by itself. This still improves performance and also minimizes the waste incurred by the splitting, so that the energy efficient execution is most effective.

10.4 Discussion and Related Work

This chapter analyzed the applicability of coscheduling in the context of the turbo boost feature available on current systems to make use of it in a more constructive manner than today's systems. With coscheduling, the schedule is modified, so that known classes of tasks can be executed with a – for them – appropriate frequency setting. As such, it creates the basis to exploit the sweet spot described in [106], which compares per-core DVFS and per-chip DVFS: get advantages similar to per-core DVFS with much cheaper per-chip DVFS.

The primary targets for this kind of coscheduling are processors capable of turbo boost, where voltage or frequency domains spans multiple cores. To this point, this includes all Intel processors with support for turbo boost, as they only have a per-chip voltage domain and also only a per-chip frequency domain. For AMD, this includes all processors with support for turbo boost – except the latest generation based on the AMD Zen microarchitecture, which was released in 2017 – as they also only have a single voltage domain. That said, even with the more fine-grained control available in AMD processors based on the Zen microarchitecture, frequency and voltage are still shared between SMT siblings. Thus, there is still opportunity to apply coscheduling to decrease energy contention – though it will inevitably double as a mean to decrease execution unit contention as well, as background load won't be able to take away as much processor cycles from more important load.

With every new generation of processors, the respective implementation of turbo boost has been improved. With Intel's Turbo Boost Technology 2.0 introduced with Sandy Bridge [107] and AMD's version of Turbo Core Technology introduced with Llano [108] the boosting became more dynamic: thermal capacity of the cooling system is considered as well as that of adjacent inactive cores. Still, to this day, all implementations consider the number of active cores as a limiting factor for attainable frequencies. Thus, unless you work on an unlocked processor or a particularly bad specimen, this is a very palpable limitation that can be exploited in coscheduling decisions. Similarly, other quirks can be exploited. For example, Intel processors up to Haswell would limit the base frequency and attainable turbo frequencies as soon as a single core would execute AVX instructions [109]. (Broadwell and later generations still have lower base frequencies for AVX workloads but are again a bit more dynamic with boosting.) This is another case, where forming classes of tasks make sense, so that AVX workloads do not slow down non-AVX workloads unnecessarily.

Intel's latest improvement in this area, *Turbo Boost Max Technology 3.0* [110], finally exposes on-chip asymmetries of cores to the operating system: only select cores are able to reach the highest turbo frequency (which is a bit beyond the advertised maximum turbo frequency). Hence, when faced with light load, the load balancer should make use of just these cores. This is mostly orthogonal to coscheduling, where – under light load – the load balancer should move coscheduled sets to those synchronization domains that contain the favored cores.

The evaluation in this chapter considered only two scheduling goals: high performance and energy efficiency. However, reality will likely be more complicated than just separating tasks into foreground and background. For instance, one could reserve turbo boost for tasks which are on a performance critical path [111], which essentially realizes a form of computational sprinting [112]. Whenever turbo boost is disabled – especially the more dynamic variants of turbo boost – the processor will regain thermal capacity for the next sprint. Another option is to have classes with different energy/performance trade-offs. This can range from a simple turbo boost on/off decision, based on whether there is a reasonable increase in performance compared to energy [113], to autotuned frequency decisions for individual applications [114], to further optimized schedules within a class with workload specific frequency adaptation [115] with a bounded loss on performance.

10.5 Chapter Notes

Parts of this chapter have been published previously by myself in [60] and [91]. While the former just outlined the idea of utilizing coscheduling to manage shared energy budgets and was restricted to present somewhat artificially generated results due to the lack of a coscheduler, the latter publication rectified this and included the results presented here.

Chapter 11

System Design Opportunities

As demonstrated in the previous chapters, coscheduling is not only useful for improving the performance of parallel applications; also sequential applications can profit from an optimized schedule enforced via coscheduling. A third application area of coscheduling is system design. This chapter explores the potential of coscheduling in that area by having a quick look at some approaches made possible by a coscheduler similar to TACO. This starts with Section 11.1 outlining an approach that allows to apply coscheduling schemes for reducing resource contention between sequential programs to parallel programs as well, which may rely on coscheduling themselves. Section 11.2 shows an adaptive locking scheme realized with coscheduling, that has the potential to improve response times compared to other locking schemes. Section 11.3 describes avenues opened by putting upper and lower limits on concurrency on certain application classes instead of using coscheduling in the traditional way. Finally, in Section 11.4 an idea is presented, how TACO can be used to replace the nowadays typically used explicit management of affinities with a more system- and user-friendly approach.

11.1 Resource Contention and Parallel Programs

Section 2.1 gave an overview of possibilities of improving a parallel application with coscheduling. Section 2.2, on the other hand, presented several ways how coscheduling can be used to improve the resource situation in a parallel system with sequential applications. These two use cases for coscheduling clash at first glance: applying the logic of algorithms to reduce resource contention at task level leads to a reshuffling of the schedule of those tasks, which likely breaks coscheduling of individual parallel applications. As shown in Chapter 8, the absence of coscheduling when an application expects to get coscheduled can lead to severe degradations in performance. On top of that, the resource usage of such an application may appear to change completely if it is not coscheduled, e. g., due to increased active waiting or a now non-functional cache optimization – independent of actual resource contention. Hence, a contention-aware scheduling algorithm has to take coscheduling requests of applications into account.

On the one hand, it is possible to trade off the benefits an application has due to coscheduling against expected benefits from reduced resource contention. Technically, this includes approaches already mentioned in Section 4.5, that autonomously decide whether parallel applications profit from being coscheduled or that merge multiple parallel applications into one coscheduled set, though especially the former are more concerned about avoiding fragmentation as a coscheduling artifact than avoiding contention caused by simultaneous execution. On the other hand, coscheduled applications can be scheduled contention-aware: constraints are kept intact and combinations of sets are coscheduled so that resource contention is minimized. Compared to work on contention-aware scheduling of sequential applications, there is surprisingly little work on contention-aware scheduling of parallel applications. There is the approach by Bhadauria and McKee [98], which also considers speed-up (besides resource contention information) to shape and select parallel applications to coschedule. After that, one has to broaden the scope and look at contention-aware placement of parallel applications [116–118] or drop preemption as a criterion [119] – approaches, which cannot be immediately re-used for a more general contention-aware scheduling.

The remainder of this section outlines a novel approach for contention-aware scheduling of parallel applications, that allows reusing contention-aware scheduling algorithms developed for sequential applications in settings with parallel applications – opening a full body of research to a new area.

Contention-aware scheduling algorithm usually base their decisions on resource statistics gathered during runtime (see Chapter 2.2 for references). Based on these statistics they decide which tasks should share or not share a particular resource at the same time. Taking coscheduling constraints as given, coscheduled parallel applications restrict the potential schedules a contention-aware algorithm may generate. At a high level, a contention-aware scheduler must now decide, which coscheduled applications should be executed in parallel, and within this set of applications it must decide which task should execute on which CPU to further minimize resource contention. In the following, these two decisions will be called *outer scheduling decision* and *inner scheduling decision*, respectively.

The inner and outer scheduling decision still depend on each other. When focusing on just one application, the optimal placement of its tasks may look completely different when it is executed with a different set of parallel programs. To fully decouple them, an artificial placement constraint – if not already present – is attached to an individual parallel application to limit it to partitions of identical shapes. Now, no matter where within the system an application is placed, any partition it receives will be equivalent in terms of shared resources to any other possible partition. Thus, the inner scheduling decision has a constant base to work with, and it can optimize task placement of a parallel application within its partition. There is no need to invalidate all gathered statistics because of a migration or a change in the application mix as advocated by the outer

scheduling decision. The "input" of the outer scheduling decision now consists of parallel applications and their statistics instead of individual tasks (and their individual statistics). Having constrained each application to resource equivalent partitions makes gathering of statistics per application feasible, as they will not inherently fluctuate due to repeated mapping changes. Instead, only behavioral changes within an application itself will cause statistics to change, after which it may take a moment for the inner decision to alleviate resource contention within the application – giving the outer decision a new normal to work with.

For both, the inner and outer decision, it is possible to employ one of the already existing contention-aware scheduling algorithms, which have proven their capability with sequential applications (cf. Chapter 2.2). While this approach – essentially a heuristic – may not be able to reach as good schedules as a more holistic approach, it cuts down the combinatoric explosion on large systems. Furthermore, with the coscheduler presented in this thesis, this approach can be easily realized without first having to develop a contention-aware scheduler specifically for parallel programs.

11.2 Adaptive Locks with Coscheduling

Traditionally, locks are either realized with active waiting or with passive waiting. If active waiting can be used efficiently (i. e., lock holder preemption is not an issue) and expected waiting times are rather short, active waiting usually results in higher performance and lower response times compared to passive waiting. Coscheduling makes active waiting efficient in multiprogrammed scenarios. However, the guarantee provided by coscheduling might be too strong for an application, restricting the operating system scheduler needlessly. Also, with coscheduled parallel applications it is only efficient to actively wait for tasks within the same parallel application.

There have been multiple approaches, trying to achieve the benefits of active waiting without resorting to a fully coordinated approach with passive waiting. First, there is spin-blocking (which coincidentally is attributed to Ousterhout [1]): if a task does not get a lock within the expected waiting time, it assumes the lock holder is currently not running and goes to sleep. The advantage of this is, that it can be fully realized in user space without operating system support. On the other hand, with a sufficiently high multiprogramming degree, nearly all spinning is pointless. In fact, it is still a research topic to find the optimal time one should spin before blocking, e. g., [120–122]. With enough meta information this may also result in implicit coscheduling [66]. Another solution, which requires operating system support, are adaptive mutexes as implemented by Solaris: a task waiting for a lock spins as long as the lock holder is running; when the lock holder is not running or gets preempted, the spinning task goes to sleep immediately [123]. This results in active waiting behavior in lightly loaded scenarios and in passive waiting behavior, when there is higher system load.

With coscheduling, and specifically its realization in this thesis, it is possible to go a step further than this. Instead of spinning needlessly or immediately falling back to passive waiting as in Solaris' adaptive mutexes, it is possible to "fall back" to coscheduling of lock holder and waiting tasks. This way, a timely handling of the protected resource is guaranteed compared to passive waiting, where it may take a while until a woken task is scheduled again. There are a few different options to utilize coscheduling for this, depending on the desired effect and on how much information is available. For example, do we know exactly which task is waiting for which other task, or do we have less information available, e. g., we only know that a certain task is waiting for something. (The latter information is provided with a slight delay by PLE/PF for virtual machines.)

One option is to associate each lock with a coscheduled set with a coscheduling constraint. This coscheduled set is empty (and hence inactive), when the lock is not taken. Whenever a task tries to take the lock, the task is added to this lock's set; when the task eventually releases the lock, it is removed again from the lock's set. (The setup of the set can be done lazily by the task encountering the taken lock to keep the uncontended case fast.) This way, whenever the lock holder is preempted, all tasks waiting for the lock will be preempted as well – similar to Solaris' adaptive mutexes. However, unlike the adaptive mutexes, the waiting tasks will start their execution again together with the lock holder instead of getting woken up only when the lock holder releases the lock. In case a lock is already taken and the lock holder is not running, a newly spinning task will either get preempted immediately or the lock holder will start executing to satisfy the set's coscheduling constraint.

From this, it is possible to avoid too many simultaneous spinning waiters by using a minimal and maximal concurrency constraint instead. Ensuring that the lock holder is always executed can be achieved for example with priorities if available, or by switching the scheduling algorithm for such a coscheduled set. If the coscheduler realization allows switching scheduling algorithms on a coscheduled set basis, using a simple FIFO algorithm for a lock's set augments ticket or queue-based spinlocks [124] at the scheduler level: the surplus of waiters is queued in the scheduler and woken in the right order when spots in the coscheduled set become available. Additionally, wakeup latencies during lock handover are avoided as this setup implicitly realizes pipelining of wakeups [121], where – depending on the width of the coscheduled set – future lock holders are dispatched ahead of time. With a different setup, more scalable synchronization constructs [125] can be supported as well.

Having just information about which tasks wait for something to happen within a group of tasks (e. g., PLE/PF in combinations with VMs), only an inferior setup can be realized, where basically all runnable tasks in the group have to be coscheduled for timely progress as the identity of the lock holder is usually unknown. However, when there is no waiter, all tasks in the group can be scheduled independently, avoiding the need for synchronization. This conditional coscheduling dodges the question which vCPU to execute next by executing all

of them in parallel instead of selecting a more or less likely candidate as other heuristics do [122].

11.3 Concurrency Management

The management strategy discussed in Chapter 9 focused on malleable applications, where applications cooperate with the operating system to decide on a good degree of parallelism. While the given management strategy is able to integrate non-malleable applications nicely, there exists a certain class of applications that – while not having a configurable degree of parallelism (DOP) – can work with a more or less arbitrary degree of concurrency (DOC). Instead of a coscheduling constraint, these applications have minimal concurrency or maximal concurrency constraints, if necessary. There is for instance the class of obstruction-free algorithms, which profit from a relatively low DOC, or the class of throughput oriented algorithms that profit from a particular high DOC (cf. [8]). Both are easily integrated into the management concept similar to malleable applications, but without the need to reconfigure their DOP on DOC changes.

In some cases, however, it is possible for an application to fully delegate the configuration of the DOP to the operating system itself – further simplifying application code and improving system utilization as reconfiguration delays are mostly avoided. The only duty that remains with an application, is to create enough tasks and signal the operating system which of them may be put on hold automatically. This concept applies naturally to worker pools, where workers process incoming work items from some kind of queue. When the operating system wants to reduce the DOP, it simply ceases execution of as many workers as necessary. If they were processing a work item at that moment, this work item will be continued when one of the still executed tasks blocks because of a lack of work (unless workers remove themselves temporarily from the set of automatically managed tasks). Increasing the DOP is done by simply continuing the execution of tasks currently on hold.

Within TACO, there are multiple ways to realize this kind of behavior. For its simplest form, it is enough to switch the scheduling strategy within a coscheduled set to a non-preemptive one. This by itself will stop and start individual tasks on DOC changes. If that is not possible, or if additional control is necessary to not stop just any task on DOC reductions, one can also use priorities, so that tasks get preempted in a well-defined order. With proper metrics, this realizes a concurrency control at the scheduler level, and can be used as the low-level mechanism to implement other approaches like [126, 127].

This concurrency management is not restricted to the aforementioned worker pools. It can also be applied to auxiliary tasks (cf. Chapter 2), where the auxiliary tasks can be put on hold as soon as the system becomes too loaded. Or it can be applied to the adaptive locks from the previous section, where it

allows to put spinning tasks on hold without stopping the lock holder itself, when there is something more important to do (which could even be idling, so that the lock holder benefits from turbo boost). Basically, this utilizes the one-sided aspect of coscheduling introduced in Chapter 3: while some tasks may only be executed together with certain other tasks, they are not required to also execute whenever the other tasks are executed.

11.4 Refined Affinity Management

One of the traditional roles of an operating system is its function as a resource manager and hardware abstraction layer. Applications do not have to care, from which components a system is assembled; they rely on the operating system to present devices in a similar, more or less abstract fashion on different systems. This separation of concerns makes applications portable and eases application development considerably. Sometimes, though, applications choose to bypass mechanisms provided by the operating system for performance reasons – an example for this (although unrelated to this thesis) are passed-through devices in virtualization scenarios. Another mechanism (and more relevant for this thesis) that certain classes of applications like to bypass, is the automatic assignment of tasks to CPUs, i. e., the management of CPU affinity. The reasons, why applications think they have to do that, are manifold.

Foremost are applications that somehow depend on specific relationship between several CPUs to achieve a maximum of performance. With the operating system not knowing about this and with no way to detect this, such an application does not realize its full potential. In this category belong, for instance, parallel applications with certain communication patterns [80,128], or SMP VMs that want to present the guest an accurate representation of the host topology. Then, there are applications which depend heavily on certain resources and want to be placed near them to reduce latency and increase bandwidth. These resources are not only I/O devices [129], but also memory in a NUMA architecture, or even a "hot" cache. Here, the goal is to prevent the operating system's load balancer from moving a task away from its resource, which is assumed to be stationary. Finally, a micromanagement of CPU affinities allows parallel applications to utilize all CPUs at a moments notice, instead of risking multiple tasks getting scheduled on one CPU and having to wait for the load balancer to detect the change in load and to rebalance tasks appropriately [130, 131].

Applications that manage their CPU affinities on their own tie themselves more closely to a certain system and are likely to need adaptations when deployed to newer, larger, or simply different systems. Additionally, such application do not integrate nicely with other applications. Consider multiple applications, which have bound their tasks to specific CPUs. When these applications have phases, where they cannot make use of the full system, they leave a fixed set of CPUs without work. This may lead to multiple applications having load on the

same CPUs while other CPUs have no load at all. And due to explicit affinities, the load balancer is unable to address that.

All these cases of explicit affinity management by applications have in common, that they address what they perceive as short-comings of the operating system's scheduler: applications (or their developers) think, that the operating system is not able to correctly infer their requirements; hence, they work around it. Whether they are right or wrong about that, does not really matter. The problem remains, that there is no interface to adequately communicate these requirements to the operating system. Upon closer inspection of the aforementioned cases of explicit affinity management, it is apparent that none of these cases needs an absolute placement of tasks. Instead, there exists a relation between multiple tasks or between tasks and resources, which makes certain placements more effective than others.

TACO provides the necessary means to convey application requirements to the operating system without being as direct as the explicit affinity management, or too broad, which would require again inferring techniques in the operating system for efficient execution. The placement constraints allow applications to specify necessary relationships between tasks or tasks and resources. A need to still bind some tasks to subsets of available CPUs does not remain. Similarly, the interface for binding memory to certain NUMA nodes can be redefined: Instead of binding memory directly to NUMA nodes, memory is attached to coscheduled sets and whether it should or must stay with the tasks within it. This gives the load balancer the possibility to still move these kinds of applications around – conjointly with their memory. If not needed for another reason, TACO's time constraints remain unused in this use case.

This refined interface for affinity management can also be used internally by the operating system itself, where for example heuristics typically used in the load balancer can now be expressed adequately. Specifically, there is the trade-off in lightly loaded systems between a more wide-spread task placement that ensures high-performance of the few tasks and a placement that ensures low energy consumption by leaving most of the system idle. With placement constraints, these heuristics are effectively represented by either an independent placement constraint or a close placement constraint in the scheduler's root set.

Even legacy applications, which still explicitly manage affinities, can get at least get some benefit from the refined interface by transparently converting from the old to the new interface. Assuming that it is known, which tasks that make use of explicit affinities constitute an application, they can be placed in a coscheduled set, which resembles the system topology from the root down to CPUs. The explicit affinities set by an application no longer reference physical CPUs, but they identify single-CPU child sets. The effect is that tasks bound to the same set will still end up on the same CPU, while the load balancer gains the freedom to permute the mapping of sets to topological units. Consider for example two parallel applications, which are both in a light phase utilizing only two of the available CPUs. Unfortunately, thanks to explicitly managed

affinities, they both utilize CPUs 0 and 1. Having affinities apply to single-CPU sets instead, the load balancer is able to move sets 0 and 1 of one application onto CPUs 2 and 3, roughly doubling the performance.

11.5 Chapter Notes

The approach described in Section 11.1 has previously been published by myself in [100]. Outside the scope of this thesis, I published an approach that automatically managed the degree of concurrency at application-level [132]. This would now heavily profit from the scheduler support described in Section 11.3.

The idea to refine the affinity management in Section 11.4 already existed as a rough sketch, when I published [84]; it is why *relocatable CPU sets* were a thing then, but are not mentioned elsewhere in this thesis. The full flexibility implied by the name and described in this chapter was never needed to implement TACO.

Chapter 12

Conclusion

Assuming that Moore's Law will hold true for a little longer, and no major technological breakthroughs come along, we are looking at a future where many more transistors can be integrated into a single processor. This transition from multicore to manycore systems – no matter how these future systems will actually look like – will put a stronger emphasis on locality again. With networks-on-chip, we will also see NUMA-like problems pop up at chip level. For software this means that there will be an enormous computational power available – but only if software goes parallel and adapts to these future systems. For example, processing user input single-threaded will become impractical, due to the ever increasing amount of data to process and the desire for a smoother user experience. Thus, we are looking at scenarios with many short parallel phases and also longer jobs for background analysis and improvement of initial results.

Given that the development of an operating system is incredibly costly, it is very likely that this manycore future will be powered by descendants of today's operating systems. Research plays an important role in this development by demonstrating the potential that can or could be unleashed even on today's systems. However, it is less often that research also offers a viable way, how this potential can be exploited in today's systems without breaking all the existing code – code that has proven over and over again that it gets the job done. For instance, have a look at my work on better operating system support for the turbo boost feature of our current microprocessors: the initial work [60], while backed by real world measurements, was basically a theoretical wouldn't-it-be-great-if construct with no possibility to quickly prototype it. The idea to turn the presented approach into something real was actually the initial spark for this thesis.

Since then, I unified the scheduling aspects of different research areas (Chapter 2) by putting them on common ground: coscheduling (Chapter 3). I developed a coscheduler design capable of supporting them all (Chapter 6). Especially, I put great effort into making sure that said design can be easily integrated into existing operating systems, without disrupting their behavior (Chapter 7). The latter is essential for acceptance by system developers and, hence, for push-

ing the development of our future operating systems forward. The rather moderate costs of my approach and the flexibility in integration, coupled with a comparatively low risk and potentially wide-spread rewards make it attractive for them to implement. A fraction of the possible gains were evaluated in Chapters 8, 9, and 10 – the latter being the practical application of the previously mentioned work on the turbo boost feature. All these have in common, that they are light-weight. That is, they can easily be applied online without further ado. Some new design opportunities for system developers, that are enabled by my coscheduler design, were discussed in Chapter 11. While they are not groundbreaking in themselves, they illustrate the power inherent to the approach. They are basically gained for free and partly clean up some longstanding issues in the operating system interface.

An implementation of the coscheduler will allow researchers in affected areas to concentrate on their core research again, while further development and optimization of the coscheduler lies mostly within the responsibility of system developers. Currently, researchers are stuck with one of the following paths if they want a practical evaluation:

- Expend resources into developing a single purpose coscheduler. This coscheduler is then able to realize just their specific use case – likely breaking scheduling behavior expected by existing workloads. Due to things considered unimportant in the research context, that coscheduler is almost surly predetermined to go extinct once research moves on.

- Use one of the larger frameworks for resource management that support coscheduling. This, however, precludes some research directions due to individual limitations of said frameworks.

- Do not use coscheduling. While this may sound counterintuitive, it actually works to some degree, because some of the studied effects are local to a processor. In a multi-socket system it is then possible to substitute separation in time with separation in space. Naturally, research options and applicability of results to actually coscheduled systems are limited here as well.

Additionally, research on energy efficiency often requires some a priori knowledge about specific properties of what is going to be executed. This knowledge is usually acquired offline. Only a few research projects actually cover, how such knowledge can be derived online. Similarly, there is the area of auto-tuning, for which the system must be exclusively available beforehand to set up everything for efficient runs. For both scenarios, the designed coscheduler provides the mean to realize a setting, where the necessary runs for task analysis and auto-tuning can be executed without external influences in a running production system. It can also ensure that results are repeatable, no matter what else is going on within the system. For these kinds of applications, coscheduling is the necessary cornerstone to catapult them from a niche existence into mainstream systems.

In the end, the availability of coscheduling – as it is promoted by this thesis – has the potential to shape the future software landscape by opening a wide range of possibilities in resource management, providing new impulses in operating system design, and by allowing to apply niche application designs on a broad scale.

Bibliography

[1] J. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS '82)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1982, pp. 22–30.

[2] OpenMP Architecture Review Board, "OpenMP application program interface, version 3.1," Jul. 2011.

[3] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[4] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306–318, 1992.

[5] F. Liu and Y. Solihin, "Understanding the behavior and implications of context switch misses," *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 4, pp. 21:1–21:28, Dec. 2010.

[6] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *Supercomputing, 2003 ACM/IEEE Conference*, Nov. 2003, pp. 55–55.

[7] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 329–342.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[9] VMware, Inc., *VMware vSphere: The CPU Scheduler in VMware ESX 4.1*, 2010, white paper.

[10] K. T. Raghavendra, S. Vaddagiri, N. Dadhania, and J. Fitzhardinge, "Paravirtualization for scalable kernel-based virtual machine (kvm)," in *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, Oct. 2012, pp. 1–5.

[11] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3*, Intel Corporation, Santa Clara, CA, USA, Sep. 2014.

[12] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Advanced Micro Devices, Inc., May 2013, rev. 3.23.

[13] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines," in *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*. Berkeley, CA, USA: USENIX Association, 2004, pp. 43–56.

[14] J. Ouyang and J. R. Lange, "Preemptable ticket spinlocks: Improving consolidated performance in the cloud," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '13. New York, NY, USA: ACM, 2013, pp. 191–200.

[15] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv—optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72.

[16] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming*. Addison-Wesley, 2004.

[17] Message Passing Interface Forum, "MPI: A message-passing interface standard, version 2.2," Sep. 2009.

[18] D. Kim, S. S.-w. Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2004, pp. 27–38.

[19] C. Jung, D. Lim, J. Lee, and D. Solihin, "Helper thread prefetching for loosely-coupled multiprocessor systems," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[20] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 269–279.

[21] J. Mars, L. Tang, and M. L. Soffa, "Directly characterizing cross core interference through contention synthesis," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC '11. New York, NY, USA: ACM, 2011, pp. 167–176.

[22] Q. Zeng, D. Wu, and P. Liu, "Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 367–377.

[23] *Software Optimization Guide for AMD Family 15h Processors*, Advanced Micro Devices, Inc., Jan. 2012, rev. 3.06.

[24] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113.

[25] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1–2, pp. 3–35, 2001.

[26] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on cmps," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07. New York, NY, USA: ACM, 2007, pp. 105–115.

[27] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:12.

[28] H. Prokop, "Cache-oblivious algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, Jun. 1999.

[29] P. J. Denning, "Thrashing: Its causes and prevention," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 915–922.

[30] A. Batat and D. Feitelson, "Gang scheduling with memory considerations," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, 2000, pp. 109–114.

[31] K. D. Ryu, N. Pachapurkar, and L. Fong, "Adaptive memory paging for efficient gang scheduling of parallel applications," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, Apr. 2004.

[32] D. Tuomenoksa and H. Siegel, "Task preloading schemes for reconfigurable parallel processing systems," *Computers, IEEE Transactions on*, vol. C-33, no. 10, pp. 895–905, Oct. 1984.

[33] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, Feb. 2005, pp. 340–351.

[34] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache pirating: Measuring the curse of the shared cache," in *Parallel Processing (ICPP), 2011 International Conference on*, Sep. 2011, pp. 165–175.

[35] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 47–58.

[36] B. Hall, M. Anand, B. Buros, M. Cilimdzic, H. Hua, J. Liu, J. MacMillan, S. Maddali, K. Madhusudanan, B. Mealey *et al.*, *POWER7 and POWER7+ Optimization and Tuning Guide*, ser. IBM Redbooks. IBM Redbooks, Nov. 2012.

[37] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *Proceedings of the 9th International Conference on Architectural support for programming languages and operating systems (ASPLOS '00)*. New York, NY, USA: ACM Press, Nov. 2000, pp. 234–244.

[38] S. Parekh, S. Eggers, H. Levy, and J. Lo, "Thread-sensitive scheduling for SMT processors," University of Washington, Tech. Rep., 2000.

[39] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012.

[40] J. Kong, S. W. Chung, and K. Skadron, "Recent thermal management techniques for microprocessors," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 13:1–13:42, Jun. 2012.

[41] A. Merkel and F. Bellosa, "Task activity vectors: a new metric for temperature-aware scheduling," in *Proceedings of the 3rd ACM*

SIGOPS/EuroSys European conference on Computer systems (EuroSys '08). New York, NY, USA: ACM Press, Apr. 2008, pp. 1–12.

[42] D. G. Feitelson and L. Rudolph, "Towards convergence in job schedulers for parallel supercomputers," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, ser. IPPS '96. London, UK, UK: Springer-Verlag, 1996, pp. 1–26.

[43] D. G. Feitelson and L. Rudolph, "Distributed hierarchical control for parallel processing," *Computer*, vol. 23, no. 5, pp. 65–77, May 1990.

[44] A. Hori, M. Maeda, Y. Ishikawa, T. Tomokiyo, and H. Konaka, "A scalable time-sharing scheduling for partitionable distributed memory parallel machines," in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 2, Jan. 1995, pp. 173–182.

[45] A. Hori, T. Yokota, Y. Ishikawa, S. Sakai, H. Konaka, M. Maeda, T. Tomokiyo, J. Nolte, H. Matsuoka, K. Okamoto, and H. Hirono, "Time space sharing scheduling and architectural support," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin Heidelberg, 1995, vol. 949, pp. 92–105.

[46] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08. New York, NY, USA: ACM, 2008, pp. 169–178.

[47] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, Apr. 2018.

[48] B. A. Braun, S. Jana, and D. Boneh, "Robust and efficient elimination of cache and timing side channels," *Computing Research Repository*, Jun. 2015.

[49] Intel, "Deep dive: Intel analysis of L1 terminal fault," Aug. 2018.

[50] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," Tech. Rep., 2018.

[51] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 973–990.

[52] T. Gleixner, "L1TF - L1 terminal fault," in *The Linux kernel user's and administrator's guide*, Jul. 2018.

[53] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, Jan. 2010, pp. 1–10.

[54] A. Hori, H. Tezuka, and Y. Ishikawa, "Overhead analysis of preemptive gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin Heidelberg, 1998, vol. 1459, pp. 217–230.

[55] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, "The network architecture of the connection machine cm-5 (extended abstract)," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '92. New York, NY, USA: ACM, 1992, pp. 272–285.

[56] Y. Peng, M. Saldana, and P. Chow, "Hardware support for broadcast and reduce in mpsoc," in *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, ser. FPL '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 144–150.

[57] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: The programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[58] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks, "A dynamic compilation framework for controlling microprocessor energy and performance," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 271–282.

[59] E. Lau, J. E. Miller, I. Choi, D. Yeung, S. Amarasinghe, and A. Agarwal, "Multicore performance optimization using partner cores," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, ser. HotPar'11. Berkeley, CA, USA: USENIX Association, 2011.

[60] J. H. Schönherr, J. Richling, M. Werner, and G. Mühl, "A scheduling approach for efficient utilization of hardware-driven frequency scaling,"

in *Workshop Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS 2010 Workshops)*, M. Beigl and F. J. Cazorla-Almeida, Eds. Berlin, Germany: VDE Verlag, Feb. 2010, pp. 367–376.

[61] *Technical Advances in the SGI UV Architecture*, Silicon Graphics International Corp., Jun. 2012, white paper.

[62] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An experiment in distributed operating system structure," *Commun. ACM*, vol. 23, no. 2, pp. 92–105, Feb. 1980.

[63] J. K. Ousterhout, "Partitioning and cooperation in a distributed multiprocessor operating system: Medusa," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, USA, Apr. 1980.

[64] J. Ousterhout, *Medusa: A Distributed Operating System*, ser. Computer Science Series. UMI Research Press, 1981.

[65] D. G. Feitelson, "Job scheduling in multiprogrammed parallel systems," IBM T. J. Watson Research Center, Research Report RC 19790 (87657), Aug. 1997, second Revision, Extended Version.

[66] A. C. Arpaci-Dusseau, "Implicit coscheduling: coordinated scheduling with implicit information in distributed systems," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 283–331, Aug. 2001.

[67] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proceedings of the 5th European conference on Computer systems (EuroSys '10)*. New York, NY, USA: ACM Press, Apr. 2010, pp. 153–166.

[68] VMware, Inc., *The CPU Scheduler in VMware vSphere 5.1*, 2013, technical whitepaper.

[69] F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M. S. Squillante, "A gang scheduling design for multiprogrammed parallel computing environments," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer Berlin Heidelberg, 1996, vol. 1162, pp. 111–125.

[70] D. G. Feitelson and L. Rudolph, "Evaluation of design choices for gang scheduling using distributed hierarchical control," *Journal of Parallel and Distributed Computing*, vol. 35, no. 1, pp. 18–34, 1996.

[71] C. H. Séquin, A. M. Despain, and D. A. Patterson, "Communication in x-tree, a modular multiprocessor system." in *ACM Annual Conference (1)*, R. H. Austing, D. M. Conti, and G. L. Engel, Eds. ACM, 1978, pp. 194–203.

[72] D. G. Feitelson, "Packing schemes for gang scheduling," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, ser. IPPS '96. London, UK, UK: Springer-Verlag, 1996, pp. 89–110.

[73] D. G. Feitelson and L. Rudolph, "Coscheduling based on runtime identification of activity working sets," *International Journal of Parallel Programming*, vol. 23, no. 2, pp. 135–160, 1995.

[74] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of OpenMP applications for multicore architectures," in *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS '10)*. Piscataway, NJ, USA: IEEE, Apr. 2010, pp. 1–10.

[75] E. Frachtenberg, D. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling: mitigating load imbalance and improving utilization of heterogeneous resources," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, Apr. 2003.

[76] E. Frachtenberg, D. Feitelson, F. Petrini, and J. Fernandez, "Adaptive parallel job scheduling with flexible coscheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 11, pp. 1066–1077, Nov. 2005.

[77] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan, "Adapt: A framework for coscheduling multithreaded programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 45:1–45:24, Jan. 2013.

[78] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 14, no. 6, pp. 581–592, Jun. 2003.

[79] J. Wang, N. Abu-Ghazaleh, and D. Ponomarev, "Controlled contention: Balancing contention and reservation in multicore application scheduling," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 946–955.

[80] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *Proceedings of the 6th European conference on Computer systems (EuroSys '11)*. New York, NY, USA: ACM Press, Apr. 2011, pp. 257–272.

[81] C. A. Waldspurger, "Lottery and stride scheduling: Flexible proportional-share resource management," MIT, Cambridge, MA, USA, Tech. Rep. MIT/LCS/TR-667, 1995.

[82] C. A. Waldspurger and W. E. Weihl, "Stride scheduling: Deterministic proportional-share resource management," MIT, Tech. Rep. MIT/LCS/TM-528, 1995.

[83] D. Giani, S. Vaddagiri, and P. Zijlstra, "The Linux scheduler, today and looking forward," *UpTimes*, vol. 2008, no. 2, pp. 41–52, 2008.

[84] J. H. Schönherr, B. Lutz, and J. Richling, "Non-intrusive coscheduling for general purpose operating systems," in *Proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT '12)*, ser. Lecture Notes in Computer Science, vol. 7303. Berlin/Heidelberg, Germany: Springer, May 2012, pp. 66–77.

[85] J. Roberson, "ULE: A modern scheduler for FreeBSD," in *Proceedings of the BSD Conference 2003 (BSDCon'03)*. USENIX, 2003, pp. 17–28.

[86] J. Roberson, "jeffr_tech's journal," Oct. 2014, http://jeffr-tech.livejournal.com.

[87] The FreeBSD Project, "SVN repository," Oct. 2014, http://svn-web.freebsd.org/base.

[88] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Secretly monopolizing the cpu without superuser privileges," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 17:1–17:18.

[89] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam, "An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. G. Feitelson and L. Rudolph, Eds. Springer Berlin Heidelberg, 2001, vol. 2221, pp. 133–158.

[90] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NASA Ames Research Center, Moffett Field, CA, USA, Tech. Rep. NAS-99-011, Oct. 1999.

[91] J. H. Schönherr, B. Lutz, J. Richling, and H.-U. Heiß, "Evaluating co-scheduling opportunities on multicore architectures," in *SJTU-TUB Workshop on Data Science and Engineering*, Mar. 2012.

[92] H. Sutter, "The free lunch is over – a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 16–22, Mar. 2005.

[93] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*. New York, NY, USA: ACM Press, Dec. 1989, pp. 159–166.

[94] C. McCann and J. Zahorjan, "Processor allocation policies for message-passing parallel computers," in *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, May 1994, pp. 19–32.

[95] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks," NASA Ames Research Center, Moffett Field, CA, USA, Tech. Rep. RNR-94-007, Mar. 1994.

[96] H. Feng, R. F. V. der Wijngaart, R. Biswas, and C. Mavriplis, "Unstructured adaptive (UA) NAS parallel benchmark, version 1.0," NASA Ames Research Center, Moffett Field, CA, USA, Tech. Rep. NAS-04-006, Jul. 2004.

[97] J. Corbalan, X. Martorell, and J. Labarta, "Improving gang scheduling through job performance analysis and malleability," in *Proceedings of the 15th International Conference on Supercomputing (ICS '01)*. New York, NY, USA: ACM, 2001, pp. 303–311.

[98] M. Bhadauria and S. A. McKee, "An approach to resource-aware co-scheduling for CMPs," in *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. New York, NY, USA: ACM, 2010, pp. 189–199.

[99] J. H. Schönherr, B. Juurlink, and J. Richling, "Topology-aware equipartitioning with coscheduling on multicore systems," in *Proceedings of the 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS-2013)*. Piscataway, NJ, USA: IEEE, Sep. 2013, pp. 1–8.

[100] J. H. Schönherr, B. Juurlink, and J. Richling, "TACO: A scheduling scheme for parallel applications on multicore architectures," *Scientific Programming*, vol. 22, no. 3, pp. 223–237, 2014.

[101] *Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors*, Intel Corporation, Santa Clara, CA, USA, Nov. 2008, white paper.

[102] *Processor Power Management in Windows Vista and Windows Server 2008*, Microsoft Corporation, Nov. 2007.

[103] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the Linux Symposium*, vol. 2, Jul. 2006, pp. 215–230.

[104] J. Richling, J. H. Schönherr, G. Mühl, and M. Werner, "Towards energy-aware multi-core scheduling," *Praxis der Informationsverarbeitung und Kommunikation (PIK)*, vol. 32, no. 2, pp. 88–95, Apr.-Jun. 2009.

[105] J. H. Schönherr, J. Richling, M. Werner, and G. Mühl, "Event-driven processor power management," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. New York, NY, USA: ACM Press, Apr. 2010, pp. 61–70.

[106] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*. New York, NY, USA: ACM Press, Aug. 2007, pp. 38–43.

[107] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, vol. 32, no. 2, pp. 20–27, Mar. 2012.

[108] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, March 2012.

[109] J. Hofmann, G. Hager, G. Wellein, and D. Fey, *An Analysis of Core- and Chip-Level Architectural Features in Four Generations of Intel Server Processors*. Cham: Springer International Publishing, 2017, pp. 294–314.

[110] Intel, "New Intel Core X series processor family," Aug. 2017.

[111] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice, "The turbo diaries: Application-controlled frequency scaling explained," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 193–204.

[112] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin, "Computational sprinting on a hardware/software testbed," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 155–166.

[113] D. Lo and C. Kozyrakis, "Dynamic management of turbomode in modern multi-core chips," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb. 2014, pp. 603–613.

[114] P. Gschwandtner, J. J. Durillo, and T. Fahringer, *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. Cham: Springer International Publishing, 2014, ch. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage, pp. 87–98.

[115] A. Merkel and F. Bellosa, "Memory-aware scheduling for energy efficiency on multicore processors," in *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower '08)*. Berkeley, CA, USA: USENIX Association, Dec. 2008.

[116] D. Goodman, G. Varisteas, and T. Harris, "Pandia: Comprehensive contention-sensitive thread placement," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 254–269.

[117] A. Collins, T. Harris, M. Cole, and C. Fensch, "Lira: Adaptive contention-aware thread placement for parallel runtime systems," in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:8.

[118] H. Zhu, L. He, B. Gao, K. Li, J. Sun, H. Chen, and K. Li, "Modelling and developing co-scheduling strategies on multicore processors," in *2015 44th International Conference on Parallel Processing*, Sept 2015, pp. 220–229.

[119] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, and P. Raghavan, "Co-scheduling algorithms for high-throughput workload execution," *Journal of Scheduling*, vol. 19, no. 6, pp. 627–640, Dec 2016.

[120] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein, "Optimal strategies for spinning and blocking," *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 246–254, 1994.

[121] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki, "A new look at the roles of spinning and blocking," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, ser. DaMoN '09. New York, NY, USA: ACM, 2009, pp. 21–26.

[122] J. Shan, X. Ding, and N. Gehani, "APPLES: Efficiently handling spin-lock synchronization on virtualized platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1811–1824, July 2017.

[123] R. McDougall and J. Mauro, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, 2nd ed. Prentice Hall, July 2006.

[124] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[125] S. Kashyap, C. Min, and T. Kim, "Scalable numa-aware blocking synchronization primitives," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 603–615.

[126] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry, "Decoupling contention management from scheduling," *SIGPLAN Not.*, vol. 45, no. 3, pp. 117–128, Mar. 2010.

[127] D. Dice, "Malthusian locks," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 314–327.

[128] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 180–186.

[129] B. Goglin and S. Moreaud, "Dodging non-uniform i/o access in hierarchical collective operations for multicore clusters," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 788–794.

[130] S. Hofmeyr, C. Iancu, and F. Blagojević, "Load balancing on speed," *SIGPLAN Not.*, vol. 45, no. 5, pp. 147–158, Jan. 2010.

[131] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–11.

[132] J. H. Schönherr, J. Richling, and H.-U. Heiß, "Dynamic teams in OpenMP," in *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '10)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2010, pp. 231–237.

Glossary

constraint refers to a restriction that is imposed on the scheduler. Any number of constraints may be attached to a coscheduled set, defining scheduling restrictions on each coscheduled set individually. There are two main types of constraints: constraints in space and constraints in time. The former control placement of scheduling entities (like a more generalized concept of affinity), the latter control simultaneous execution of scheduling entities (like coscheduling or other forms of controlling the degree of concurrency).

coscheduled set refers to a set of tasks or nested coscheduled sets and a set of associated constraints. The scheduler is responsible for placing and executing coscheduled sets, honoring their constraints.

coscheduling in its most general meaning refers to the conscious simultaneous execution of selected tasks. There have been many variations on the exact meaning of coscheduling since the introduction of this term in 1982 and related terms, like gang scheduling. This thesis sticks to the term coscheduling in its most general meaning unless specified otherwise.

degree of concurrency refers to the number of running tasks within a coscheduled set (as opposed to the degree of parallelism, which refers to the number of runnable tasks). Constraints attached to a coscheduled set directly influence the degree of concurrency.

scheduling domain refers to a concept within Linux used by the load balancer. Coincidentally, it is closely related to the concept synchronization domains in this thesis, even sharing the same abbreviation. A scheduling domain typically encapsulates a topological unit within the system, for which the load balancer gathers statistics in order to move tasks between children of the topological unit.

scheduling entity is an entity whose execution is controlled by the scheduler. Traditionally, that has been a task – a single thread of execution. Linux as well as this thesis need this additional abstraction as other types of scheduling entities are introduced. Linux uses a task group scheduling entity (TG-SE) to refer to a set of SEs on the same CPU in order to achieve fairness at group level as opposed to fairness at task level. This

thesis introduces a synchronization domain scheduling entity (SD-SEs) to refer to a set of TG-SEs across multiple CPUs, which are to be coscheduled.

synchronization domain refers to a part of the system, typically a topological unit, for which the scheduling decision is coordinated across the children of the topological unit, i. e., where we will see a collective context switch. The number and size of synchronization domains varies over time depending on the currently executed coscheduled sets.

task group refers to the concept in Linux to form a set of tasks, which is then used to achieve fairness at group level as opposed to fairness at task level. In addition to Linux' regular task groups, which are only about accounting, this thesis introduces scheduled task groups, which are an implementation of coscheduled sets.

topological unit is a coherent architectural part within a computer system, like a CPU, a processor core, a processor, or a NUMA node. The system topology of current systems can usually be described by a tree of topological units, where each topological unit at one level consists of multiple topological units of the next level, interconnected in some way and potentially with some dedicated resources. This abstraction keeps various definitions independent of the concrete computer architecture.

topology-aware coscheduling is the method introduced and used in this thesis to achieve coscheduling. Compared to other approaches, it puts a stronger focus on system topology to achieve scalability. It is also modular in the sense that each coscheduled set can have its own, independent set of constraints, making it a suitable method for many coscheduling use cases even at the same time on the same system.

Acronyms

CFS Completely Fair Scheduler. 89

cgroup control group. 90

CPU central processing unit.

DOC degree of concurrency. 38, 155

DOP degree of parallelism. 38, 155

DVFS dynamic voltage/frequency scaling. 32, 142

HPC high performance computing.

IPI inter processor interrupt.

KVM kernel-based virtual machine.

LHP lock holder preemption. 17

NUMA non-uniform memory access.

OS operating system.

PF pause filtering. 17, 154

PLE pause loop exiting. 17, 154

PV paravirtualization. 17

RTG regular task group. 94

SD synchonization domain. 67. scheduling domain. 92.

SE scheduling entity. 36, 86, 90

SG scheduling group. 92

SMT simultaneous multithreading.

STG scheduled task group. 94

TACO Topology-Aware Coscheduling. 67

TU topological unit. 37, 76

vCPU virtual CPU.

VM virtual machine.