Karlsruher Institut für Technologie

Task Activity Vectors: A Novel Metric for Temperature-Aware and Energy-Efficient Scheduling

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik
des Karlsruher Institutes für Technologie (KIT)

genehmigte

Dissertation

von

Andreas Merkel

aus Karlsruhe

Tag der mündlichen Prüfung:	4. Februar 2010
Erster Gutachter:	Prof. Dr. Frank Bellosa
Zweiter Gutachter:	Prof. Dr. Wolfgang Karl

Abstract

Over the past decades, microprocessors have seen an enormous increase in integration density and power consumption. Today, we have reached a point where on-chip temperature has become a severe problem, and energy efficiency is of paramount importance. In addition, increasing the frequency or the complexity of a processor core is no longer economical. This has lead to the introduction of explicit thread-level parallelism on processor chips (simultaneous multithreading, chip multiprocessing). Threads running in parallel on a chip compete for shared chip resources and are affected by chip-wide power management, which leads to interdependencies between those threads.

All three aspects—temperature, energy efficiency, and interdependency between threads of execution—are strongly connected to the characteristics of the applications (tasks) executed by the processor. Different tasks utilize chip-related resources such as integer or floating point units, caches, or the memory interface to different degrees. Since power dissipation is caused by activity on the chip, processor temperature is inherently coupled to the utilization of the resources on the chip and thus depends on the running task. Tasks that run in parallel and utilize the same shared chip resources to a high degree lead to contention, poor performance, and poor energy efficiency.

The operating system scheduler—managing the running tasks—can take great influence on temperature and resource contention by virtue of its scheduling decisions, i.e., by deciding which tasks to run at what time and in combination with which other tasks. Schedulers found in today's general purpose operating systems are unaware of the utilization of chip resources caused by the tasks they manage. Uninformed scheduling decisions lead to thermal problems, resource contention and, overall, inefficient use of the processor's resources.

In this thesis, we propose *task activity vectors* as a new metric to guide scheduling. An activity vector provides the scheduler with information about which processor resources a particular task uses. With this knowledge, the scheduler can make more informed scheduling decisions and use the chip's resources more efficiently.

Two use cases demonstrate the applicability of task activity vectors. Firstly, we show that the information provided by activity vectors can be used to attain a more balanced temperature distribution on the chip and to avoid pernicious hotspots by controlling the temporal order in which tasks are scheduled. Secondly, we show that activity vectors are suitable for co-scheduling tasks in a way that avoids resource contention and improves performance as well as energy efficiency.

We implemented our proposed policies for the Linux kernel. Our evaluations with multithreaded and multicore processors show that vector-based scheduling is a low-overhead way to improve performance, attain better thermal behavior, and make more efficient use of energy.

Contents

1 Introduction

1.1 Modern Microprocessors—a Challenge for Task Scheduling

In the past decades, microprocessors have seen a steady increase in integration density and power consumption. This has lead to the following situation:

- Heat has become a severe problem. Today's processor chips dissipate power in the order of 100W in an area as small as $1cm^2$, which is ten times the power density of a hot-plate. This necessitates considerable cooling efforts, as well as thermal management features provided by the hardware to counteract thermal emergencies. Thermal management has implications on the computational performance of the chip, because preventing thermal emergencies usually means slowing down the processor. Since chip temperature depends on the instructions executed by the chip and thus on the software, achieving optimal performance under thermal constraints necessitates software that is aware of thermal problems and supports thermal management.

- Energy efficiency is becoming an ever more important issue. Better energy efficiency mitigates thermal problems and reduces the need for cooling. In addition, rising energy costs increase the demand for energy efficient computer systems in the area of compute centers as well as for workstations. For mobile computers, energy efficiency enables longer battery times. This has led to the inclusion of power management features in today's microprocessors. Similar to thermal management, the power management mechanisms offered by the hardware affect performance, and the efficiency of the mechanisms at conserving energy depends on the instructions executed by the chip and thus on the software.

- Increasing the operating frequency or the complexity of a processor core in order to increase performance is no longer economical. Instead, thread-level parallelism has been introduced in today's processors in the form of simultaneous multithreading (SMT) or chip multiprocessing (CMP). In contrast to traditional symmetric multiprocessing with physically distinct chips, SMT or CMP threads share resources of one chip and are thus likely to influence each other. In particular, if the software running on the individual hardware threads utilizes the same shared resources excessively, contention can lead to poor performance and bad

energy efficiency. In addition, power management features such as frequency scaling often affect all threads running on a chip, which creates further interdependencies.

In this thesis, we address the problems of temperature, energy efficiency, and interdependency between execution contexts (that is, SMT threads or CMP cores) by adapting *operating system task scheduling*, i.e., the assignment of running applications to processors performed by the operating system.

The operating system is the central component that is suitable to address the mentioned problems from the software side. In its role as a resource manager, the operating system has knowledge about and exercises control over both hardware as well as applications. Task scheduling is a central functionality of an operating system and has great implications on temperature and energy efficiency, since the task selected for running on a processor determines the processor's power consumption and temperature [BWWK03, IM03]. In SMT and CMP systems, contention for shared resources makes co-scheduling, i.e., selecting what tasks run in parallel on the hardware contexts, decisive for performance and energy efficiency, since the combination of tasks determines the degree of resource contention [ST00, BP04, MM07].

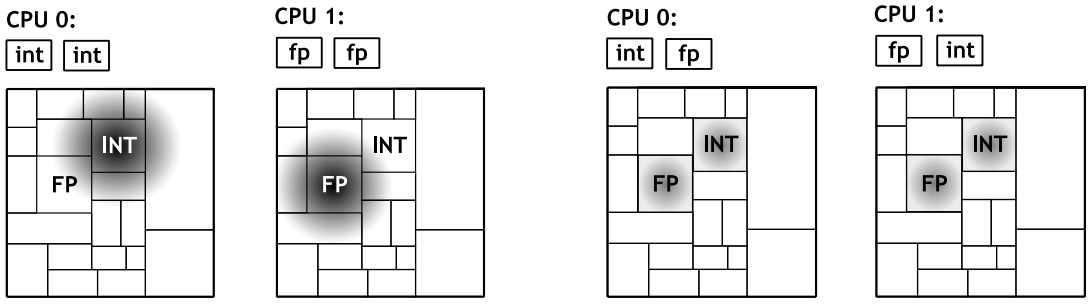
We argue that in order to address the problems of today's processors, a scheduler needs to be aware of task specific characteristics like the utilization of processor-related resources. Only with this knowledge, a suitable scheduling policy can consider the implications that scheduling decisions have on temperature, energy efficiency, and on the interdependencies between tasks running in parallel.

To limit the complexity of our research, the focus of this thesis is on single-threaded applications. In the remainder of the thesis, when we use the term "task" we will therefore imply that a task is single-threaded unless mentioned otherwise.

1.2 The Role of Resource Utilization

1.2.1 Temperature-aware task scheduling

Heat is a localized phenomenon caused by switching activity in different parts of the chip. Processor chips encompass various components such as functional units (arithmetic logic units (ALUs), floating point units (FPUs), branch predictors, reorder buffers), storage units (register files and caches), or interface units (cache and memory interface). Different tasks utilize these resources to varying degrees. Therefore, the switching activity in the chip units and the temperature distribution on the chip depend on the task being executed. High utilization of a particular resource can lead to an increased temperature in the respective part of the chip, resulting in a *hotspot*. To avoid damage to the chip, thermal management reduces the chip's power consumption by *throttling*, implemented with mechanisms that reduce switching activity but also performance.



(a) Inauspicious scheduling: Tasks utilizing the same resources on a processor cause hotspots.

(b) Auspicious scheduling: Tasks utilizing different resources on a processor lead to a more balanced temperature distribution.

Figure 1.1: Impact of scheduling on temperature distribution

The decisions that a scheduler makes have a direct impact on temperature distribution and thus indirectly affect throttling and performance. Depending on the current temperature distribution on the chip and the resource utilization of the selected task, a scheduling decision can either aggravate thermal problems, if a task is selected that utilizes resources that already have an increased temperature, or mitigate thermal problems if a task is selected that utilized resources having only a moderate temperature.

Figure 1.1 shows an example of a dual-processor system with two floating point tasks and two integer tasks. A scheduler that is unaware of task characteristics could possibly schedule both integer tasks on CPU 0 and both floating point tasks on CPU 1, causing continuous activity in the respective units of the CPUs (left part of the figure). Thus, the integer unit would become a hotspot on CPU 0, while the floating point unit would become a hotspot on CPU 1. Eventually, thermal management would have to intervene and reduce temperature by throttling at the cost of performance penalties.

A scheduler that is aware of the tasks' resource utilizations and their implication on temperature can schedule one integer task and one floating point task alternatingly on each CPU, allowing integer and floating point units to cool down during the periods of inactivity (right side of Figure 1.1), and rendering throttling unnecessary. *Knowledge about the resource utilization of tasks enables temperature-aware scheduling policies that lead to a balanced temperature distribution and avoid overheating parts of the chip.*

1.2.2 Co-scheduling for energy efficiency

For SMT chips, almost all chip resources are shared by multiple logical processors. To a lesser extent, this is also the case for CMP chips, where memory access infrastructures and, in some cases, caches are shared by multiple cores residing on a chip.

The resource utilization characteristics of tasks running in parallel on different logical processors or cores determine the degree of contention for shared resources and thus the performance. For example, the demands of several memory-bound tasks running in parallel on a multicore chip can easily exceed the memory bandwidth available to the chip. This leads to reduced performance because of stall cycles introduced on the cores while waiting for memory requests to be serviced.

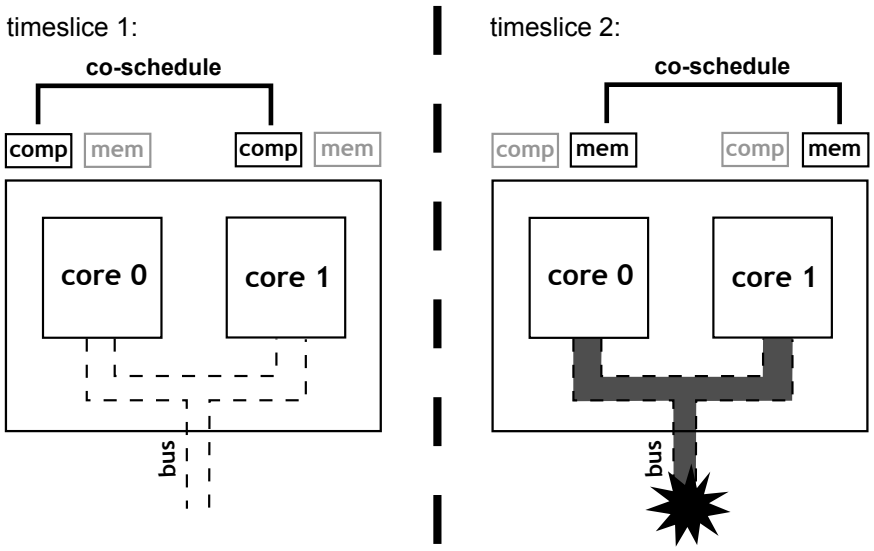
Figure 1.2 shows a dual-core system with two compute-bound tasks and two memory-bound tasks. Scheduling both compute-bound tasks during one timeslice and both memory-bound tasks during the next timeslice (top half of the figure) leads to unused capacity of the memory bus in the first timeslice and a bottleneck with corresponding performance degradation for the memory-bound tasks in the second timeslice. This performance degradation is avoided if memory-bound tasks are co-scheduled with compute-bound tasks each timeslice, making optimal use of the memory bus's capacity (bottom half of Figure 1.2).

A scheduler that is unaware of resource sharing will inadvertently schedule tasks in a way that causes resource contention and sub-optimal performance. Sub-optimal performance in this context also implies inefficiency in terms of energy: The power consumed by the processor is wasted if the processor has to stall while waiting for resources to become available. *Knowledge about the resource utilization of tasks enables co-scheduling policies that avoid contention and increase performance and energy efficiency.*

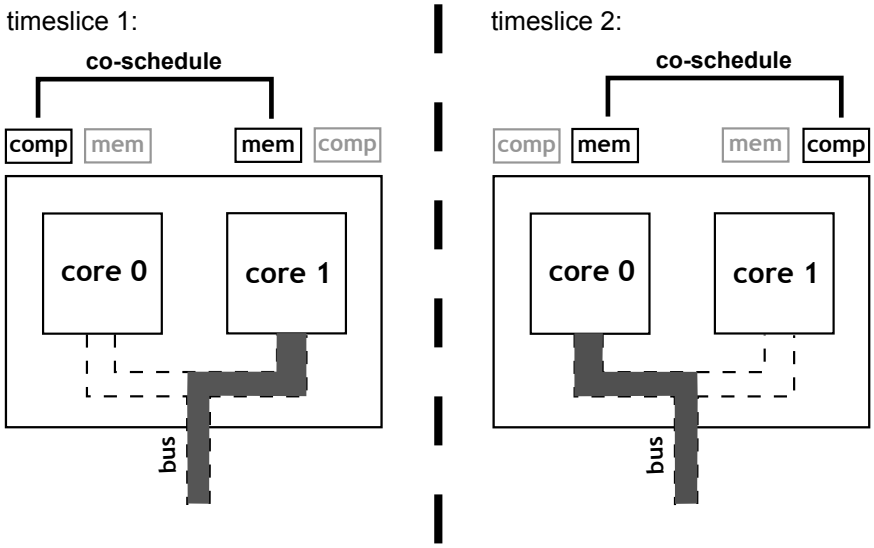
Power management features such as frequency and voltage scaling create further interdependencies between the cores of a CMP, which need be taken into account for optimal performance and energy efficiency. Many multicore chips offer frequency and voltage scaling only chip-wide, meaning that all cores need to run at the same frequency and voltage, since allowing multiple frequencies and voltages introduces additional hardware complexity.

The optimal frequency at which the processor can execute a task most efficiently in terms of runtime and energy depends on the task's characteristics [WB02, CSP04, KDG+04], in particular on the frequency of memory accesses. With chip-wide frequency scaling, it is only possible to run all tasks at their corresponding optimal frequency if tasks with similar characteristics are running on all cores of a chip. This however, can be in conflict with the goal of avoiding contention, since tasks with similar characteristics are likely to utilize the same resources.

As a consequence, when taking both, resource contention and frequency selection into account, *the question arises whether it is more advantageous to run tasks with different characteristics or tasks with similar characteristics together*, since the former avoids resource contention, while the latter allows to run the chip at a frequency that is optimal for all tasks.



(a) Inauspicious co-scheduling: tasks utilizing shared chip resources simultaneously cause contention.



(b) Auspicious co-scheduling: tasks utilizing shared resources at different times avoid contention.

Figure 1.2: Impact of scheduling on resource contention

1.3 Contributions

A scheduler that is supposed to exercise control over temperature distribution or resource contention needs information about the tasks it manages. We show that information about the utilization of processor resources can be used as a basis for scheduling policies that attain an improved temperature distribution, reduced resource contention, and better energy efficiency. Such information is not available to schedulers in today's operating systems, so these schedulers make resource-unaware decisions leading to thermal problems, contention, and energy inefficiency.

Our thesis makes three main contributions: *task activity vectors* as an abstraction to represent resource utilization, *temperature-aware, vector-based scheduling* as a policy to mitigate thermal problems and *resource-conscious scheduling and frequency selection* as a policy to achieve energy efficiency.

Task activity vectors We introduce the concept of *task activity vectors* as an abstraction that represents a task's resource utilization. An activity vector is part of the task's runtime context and describes the degree of utilization the task causes for various chip-related resources when executed. This enables scheduling policies to consider resource utilization when selecting a task to be scheduled.

We show that the operating system can maintain task activity vectors with minimal overhead by using performance monitoring counters, which are present in almost all modern processors.

Temperature-aware, vector-based scheduling Based on activity vectors, we propose scheduling policies that strive at balancing the temperature within a chip and between chips to avoid hotspots and throttling. We achieve this (1) by scheduling tasks that use different resources successively on a chip, (2) by distributing tasks among the processors of a multiprocessor system so that each processor executes tasks that use different resources, and (3) by running tasks that use different resources simultaneously on multithreaded processors. Our evaluations show that vector-based scheduling succeeds at reducing hotspots considerably.

Resource-conscious scheduling and frequency selection We analyze what is the optimal way to co-schedule tasks considering the criteria of resource contention and frequency selection. We find that in order to optimize the product of runtime and expended energy (energy delay product, EDP) in today's architectures, the main goal must be to avoid contention by combining tasks that use different resources; it is not worthwhile to co-schedule tasks that share a common optimal frequency if this would lead to resource contention.

According to our analysis, we propose two scheduling policies that avoid resource contention by co-scheduling tasks with different resource demands, based on the in-

formation provided by activity vectors. We propose a specialized policy applicable if there is one resource mainly responsible for contention, and a generic policy that is applicable if different resources cause contention. For workloads that contain too many memory-bound tasks for our scheduling policies to avoid contention, we propose a heuristic that engages frequency scaling as a fallback to mitigate the waste of energy introduced by memory contention. Our evaluation with a Linux implementation of vector-based co-scheduling reveals that our policies manage to reduce EDP considerably.

1.4 Structure

The rest of this thesis is structured as follows: Chapter 2 discusses the background upon which our work depends. In Chapter 3, we introduce the abstraction of task activity vectors. We cover the application of activity vectors to scheduling in Chapters 4 and 5. Chapter 4 shows how vector-based scheduling can tackle the problem of thermal imbalances and hotspots, and leads to a better and more balanced temperature distribution. Chapter 5 shows how the information provided by activity vectors can be used to reduce resource contention and to make efficient use of a microprocessor in terms of time and energy. With Chapter 6, we conclude and discuss directions for future work.

2 Background

In the following, we briefly introduce the terminology we will be using in the remainder of the thesis. We also cover common hardware and operating system principles that constitute the foundation of our thesis.

2.1 Tasks

We have already introduced the term *task* for a running application. More precisely, a *task* is an abstraction that encompasses flows of execution (*threads*), a view on memory protected from other tasks (*address space*) and other resources managed by the operating system such as open files or communication channels. As mentioned, we will use the term *task* in the meaning of "single-threaded task".

2.2 Symmetric Multiprocessing

In a multiprocessor system, multiple processors operate in parallel. The focus of our thesis lies on symmetric, cache coherent, shared memory multiprocessor systems. In a *shared memory* multiprocessor system, all processors have access to a shared memory for storing data and code. *Cache* is fast processor-local memory of limited size that buffers contents of main memory in order to hide memory latencies. A *cache coherent* multiprocessor system provides consistency of data written to or read from different processors' caches. In a *symmetric multiprocessor system*, all processors are equal, i.e., identical pieces of hardware. From the software side, symmetric multiprocessing means that the operating system as well as any application can run on any processor. Originally, symmetric multiprocessor (SMP) system were assembled of several physically different single-core, single threaded processor chips. We will refer to such systems as *traditional SMP* systems.

Processors of an SMP system communicate with memory via a shared memory interconnect. This way, limited memory bandwidth can become a bottleneck, especially in systems with many processors. To mitigate contention for the memory interconnect, *non uniform memory access* (NUMA) systems partition their processors into *nodes*. Each node possesses its own memory and memory interconnect. Access to memory of other nodes is possible, but has a higher latency than access to node-local memory.

2.3 Chip Multiprocessing and Simultaneous Multithreading

A *chip multiprocessor* (CMP) consists of several processors, also termed *cores*, laid out next to each other on one chip [ONH+96]. Like traditional SMP systems, the cores of a CMP share a common memory, potentially with cores from other chips if the system consists of multiple CMPs. Depending on the cache architecture, cores of a CMP can also share a common cache.

Simultaneous multithreading (SMT) means that a processor has multiple execution contexts called *logical processors*, which are able to simultaneously issue instructions from different flows of execution (e.g., different tasks) to the chip's functional units [TEL95]. To the software, the logical processors look like independent processors, each with its own registers and state. On the hardware side, the logical processors share the resources (e.g., ALUs, FPUs, caches) of one single processor. For distinction from the logical processors, we call this the *physical processor*. Logical processors of the same physical processor are termed *siblings*.

2.4 Multiprocessor Scheduling

In an SMP system, a task can run on any processor. If there are more runnable tasks than processors in the system, next to the question which task to execute on which processor, the additional question arises which tasks shall be executed on a processor at a certain point in time and which ones not. In its role as a resource manager, the operating system performs *scheduling*, the assignment of tasks to processors. The corresponding component of the operating system is called *scheduler*.

Schedulers found in general purpose operating systems like Linux or Windows perform *time-sharing multitasking*: They assign a processor to different tasks in a row for a defined period of time, a *timeslice*. This way, several tasks can make progress and the user has the impression that the tasks are running in parallel, even if there are more tasks than processors.

For SMP systems, most of today's general purpose operating systems do *affinity scheduling*, which means they associate each task to a particular processor and run the task—either preferably or exclusively—on this processor. This takes advantage of the fact that a task that has once run on a particular processor has built up some state; for example, the data the task is working on is in the processor's cache [SL93].

Many operating systems, like recent versions of Linux, partition the set of runnable tasks between the processors and assign a subset of tasks to each processor. The scheduler manages the set of runnable tasks assigned to a processor in a data structure, the *runqueue*, which keeps track of all tasks eligible for running on a specific processor. To provide optimal performance and fairness, schedulers perform *load balancing*; they

migrate tasks between runqueues for equalizing the number of tasks assigned to each processor.

If scheduling decisions on different processors happen in a coordinated fashion, and tasks are selected to run in parallel on different processors following a defined scheme, we speak of *co-scheduling*. In today's general purpose operating systems, co-scheduling is usually not performed.

Most general purpose operating systems schedule the tasks of a runqueue following the same basic principles. Firstly, all tasks are supposed to make progress. The simplest scheduling policy that ensures this property is *round robin*, i.e., executing each task in turn for a timeslice. Secondly, most operating systems allow a form of prioritization in the sense that higher priority tasks are allotted more CPU time. Thirdly, there is often a distinction between *interactive* and *CPU-bound* tasks. Interactive tasks tend to use the processor only for short periods of time and block frequently, waiting for input data from a user or a peripheral device like a hard disk. CPU-bound task occupy the processor continuously and have little interaction with the user or peripheral devices. Therefore, many schedulers favor interactive tasks over CPU-bound tasks to give the user the impression of a more responsive system and to utilize peripheral devices better.

2.5 Thermal Behavior of a Processor Chip

The units a microprocessor consists of (ALUs, FPUs, caches, register files, and so on) are usually laid out as blocks, i.e., (mostly) rectangular areas on the die. Figure 2.1 shows the floorplan of an Intel Pentium 4 Northwood, derived from a die photo, as an example.

From a physical point of view, the units are located on a silicon die, which in today's processor generations is of rather small dimensions—typically in the order of one centimeter in width and length and of one millimeter in height. The die is covered by a heat spreader consisting of thermally well conducting material and by a heat sink, which is comparably large in relation to the die and consists of copper or aluminum. In relation to its size, the die dissipates large amounts of power, which can surpass 100W in recent processors. This results in a power density of 100W/cm^2 and more.

The temperature of an object, be it a chip unit, the die as a whole, or the heat sink, is determined by the amount of energy that the object contains (referred to as *internal* or *intrinsic energy*) and its *heat capacity*. The heat capacity is a material specific constant describing how many Joules it requires to heat the object by one Kelvin.

The heat capacity of the die is small compared to its power dissipation. Therefore the die (and thus the individual chip units) can heat up rather quickly. A small example shall demonstrate this: A silicon die of 11mm$\times$12mm$\times$0.7mm (which corresponds to the chip we use in our evaluation of temperature-aware scheduling in Chapter 4) has a heat capacity of 0.15J/K. If it dissipates 70W and no heat is removed from the die,

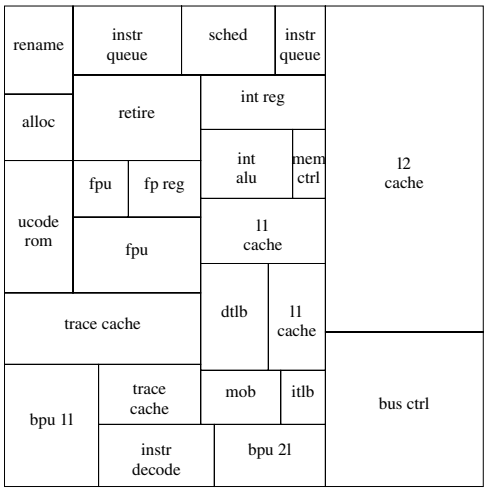


Figure 2.1: Floorplan of the Intel Pentium 4 Northwood processor

it takes only 2ms for the die to heat up by 1K. In contrast to that, the heat sink has a much larger heat capacity, since it is considerably more massy, and typically takes seconds to heat by 1K.

A rapid increase in chip temperature as described in the example happens primarily when there is a boost in chip power consumption, since the removal of heat via conduction requires a temperature gradient between the chip and the heat sink having built up in the first place. The bigger the gradient, the more heat is removed; therefore the increase in temperature slows down, until temperature settles at a certain level, depending on the power consumption (see Figure 2.2). The inverse happens when the chip's power consumption drops.

Since different chip units have different structures and are—depending on the software—utilized to varying degrees, they show mutually different power densities. In addition, owing to the small height of the chip, lateral heat conduction between neighboring chip areas is limited. As a result, the temperature varies from chip unit to chip unit, so there typically is a non-uniform temperature distribution on the chip.

The course of the intrinsic energy and the temperature of the units on the die and of the heat sink can be described by similar differential equations. The solution to those equations is an exponential function [BWWK03, MB06]. The function

$$\vartheta(t) = \frac{-\tilde{c}}{c_2} \cdot e^{-c_2 t} + \frac{c_1}{c_2} \cdot P + \vartheta_0 \tag{2.1}$$

describes the course of temperature for the heat sink or of a single chip unit (neglecting lateral heat conduction to other units) over time. The only difference between the heat

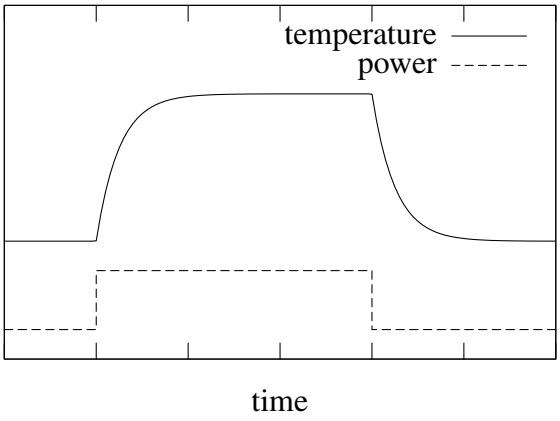


Figure 2.2: Dependence of temperature from power

sink and the chip units is the size of the constants in the equation. The constants c_1 and c_2 depend on the thermal resistance and the thermal capacitance of the chip units or the heat sink; $\tilde{c}$ is an integration constant depending on the initial temperature at the time $t = 0$. ϑ_0 is the temperature of the component the heat is conducted to (i.e., ambient air or heat sink).

2.6 Thermal Management

With increasing power densities, designing cooling facilities for the theoretical maximum power that a processor can consume in the worst case results in over-provisioning and high costs. Thus, cooling is usually designed for a lower *thermal design power* (TDP). If the processor exceeds the TDP, overheating and damage to the chip can result. Preventing this requires *thermal management*, i.e., monitoring temperature and engaging countermeasures in case of critical temperatures.

For thermal management, processors support hardware mechanisms that decrease power consumption, for instance by scaling the voltage and/or frequency, by modulating the clock signal, or by introducing halt cycles [BM01, SSH+03]. We refer to these mechanisms as *throttling*. All throttling mechanism have in common that they not only reduce power consumption, but also performance.

Since thermal management must reliably prevent thermal emergencies, policies for thermal management are typically implemented in hardware or firmware. Examples are Intel's Catastrophic Shutdown Detectors (Thermal Monitor 1 and 2), which engage clock modulation or frequency scaling if a certain temperature threshold is reached [Int06].

2.7 Power Management

Similar to thermal management, *power management* uses hardware mechanisms to reduce power consumption in exchange for reduced performance. However, the focus of power management is not limited to preventing thermal emergencies. The aim of most power management policies is to make efficient use of energy, for example, to perform a given task using as little energy as possible. This leads to longer battery times on mobile systems, or lower expenses for power and cooling in data centers.

For processor power management, frequency and voltage scaling is often the mechanism of choice, since it allows non-linear power reduction in relation to the runtime increase it causes. For most other throttling mechanisms, the increase in runtime is proportional to the reduction in power. In that case, consuming less power, but for a longer time, does not lead to energy savings.

Using a very simple model, the power dissipation of a processor can be described as proportional to the square of its operating voltage and as proportional to its operating frequency [BB95, Fle01, Mud01]:

$$P \propto V^2 \cdot f$$

The voltage required to drive the processor reliably, in turn, is (within a limited range) approximately proportional to the operating frequency:

$$V \propto f$$

Scaling down the voltage requires a low enough frequency, so voltage scaling is commonly applied in combination with frequency scaling (*dynamic voltage and frequency scaling*, DVFS). With frequency and voltage scaling in combination, the power is proportional to the cube of the frequency (combination of the above two equations):

$$P \propto f^3$$

Since performance is in the best case proportional to the frequency, DVFS allows net energy savings (it reduces power by a greater factor than it increases runtime). Although, in practice, these savings are diminished by static contributions to power consumption such as leakage, DVFS is beneficial in certain scenarios, for example, when not the speed of the processor, but the speed of other components such as memory is decisive for performance [WB02, CSP04].

The simplest option for implementing DVFS for multicore hardware is *per-chip DVFS*, i.e., running all cores at the same frequency and voltage. Adding multiple clock generators and latches for synchronizing the data flow between different clock domains allows running cores at different frequencies (*per-core DVFS*). However, without additional voltage regulators (which introduce additional complexity), all cores need to run at the voltage determined by the core running at the highest frequency,

which greatly diminishes the potential for energy savings. Recently, on-chip voltage regulators have been proposed for enabling per-core DVFS [KGWB08]. Though on-chip regulators facilitate running different cores at different voltages, introducing multiple voltage domains still causes overhead in terms of die area and energy efficiency.

2.8 Energy Efficiency Metrics

The mechanism of DVFS allows the processor to execute tasks while requiring less energy, at the expense of a prolonged runtime. For situations where both performance and energy consumption are important, the *energy delay product* (EDP) has been proposed as a metric [HIG94, GH96]. The EDP is obtained by multiplying the runtime of a task with the amount of energy required for the run. A similar metric that gives more weight to runtime is ED^2, the product of energy and squared runtime.

2.9 Event Monitoring Counters

Event monitoring counters are model-specific registers that are able to count various processor-internal events. Event monitoring counters found in today's commercially available processors were introduced for performance analysis and profiling (*performance monitoring counters*) and are therefore tailored to performance-critical events like, for example, cache misses or mispredicted branches.

Despite their focus on performance, in the past, event monitoring counters of various architectures have also been used for inferring the activity of different parts of the processor chip [IM03] and for deducing power consumption and temperature [BWWK03].

3 Task Activity Vectors

Information about what chip resources particular task is using constitutes valuable input for a scheduler. In order to make informed decisions concerning which tasks to run at what time and in which combination, a scheduler needs to know what effects running a particular task or a particular combination of tasks will have; it needs to be provided with a characterization of the tasks it manages.

In this chapter, we introduce the abstraction of the *task activity vector* as a way to characterize a task by the resource utilization it causes [MB08b]. The notion of *vector-based scheduling* describes a scheduling policy that makes use of the task characterization provided by activity vectors to attain a specific goal.

3.1 The Need for Task Characterization

As outlined in Chapter 2, today's general purpose operating systems like Linux or Windows categorize tasks by applying user-specified priorities, and, in addition, distinguish between interactive and CPU-bound tasks. Still, CPU-bound tasks having the same priority are treated as equal. We argue that treating such tasks as equal for scheduling is no longer optimal on today's processors, and that a coarse categorization into interactive and CPU-bound tasks does not consider the importance of the utilization of CPU-related resources.

In order to make optimal decisions, a scheduler needs to know what consequences running a particular task will have, for instance, what will be the impact on chip temperature and temperature distribution, and how the task will interact with other tasks running on the same chip. Thus, we need to provide the scheduler with a way to judge the impact of running a task.

As mentioned in the introduction, phenomena like the dissipation of energy and the interaction between tasks running on a chip are closely coupled to the utilization of processor-related resources like the units on the chip (e.g., ALUs, FPUs, register files, or caches) and the memory interconnect. A task that causes a high utilization of a particular resource leads to an increased temperature in the corresponding part of the chip because of increased switching activity. If the resource in question is shared between several logical processors or cores, the task will interfere with tasks running on other logical processors or cores. If several tasks utilize a resource at the same time, this leads to a slowdown of the tasks involved, and, in turn, to inefficient use of energy.

We propose the concept of task activity vectors to reflect the growing importance that the characteristics of tasks have on the temperature, the performance, and the energy efficiency of modern processors. The activity vector is an abstraction that models the resource utilization caused by a task. Task activity vectors make the resources the task uses part of the task's runtime context, so the operating system and especially the scheduler have detailed information about the characteristics of each task and can foresee and consider the implications that running this particular task will have.

3.2 Definition of Task Activity Vectors

We define a *task activity vector* as an n-dimensional vector that is part of a task's runtime context. The dimension of an activity vector corresponds to the number of resources we want to consider, and each component of the vector corresponds to the degree of utilization of one specific resource. The values of the vector's components range between 0 (the resource is not utilized at all) and 1 (the resource is fully utilized).

We define utilization as the number of accesses a task makes to the resource in a period of time, divided by the maximum number of accesses the resource supports in that period of time. The maximum can either be calculated theoretically by studying the microarchitecture (for instance, deducing that an integer unit containing three ALUs can execute up to three integer operations per cycle) or measured using specific microbenchmarks (for instance measuring memory bandwidth with the `stream` [McC95] benchmark).

For chips that support multiple operating frequencies, we define the activity vector of a task by the resource utilization the task causes at the chip's maximum frequency. We postpone a detailed discussion of activity vectors in connection with frequency scaling to Chapter 5.

In general, an activity vector can be used to model any resource that can be utilized by a task to varying degrees, including, for example, network or disk bandwidth. In this thesis, we focus on resources related to the processor chip, since we want to study the implications of scheduling on processor-related phenomena. Including the utilization of other resources such as peripheral devices is a topic for future research.

For most resources, the definition of utilization by access frequency is a suitable, and the only sensible way to define utilization. An exception are storage units such as cache or memory, for which utilization can also be defined in terms of space occupancy. For example, two memory-bound applications running in parallel on a CMP are competing for both memory bandwidth and for memory space, and performance is diminished both by insufficient bandwidth (stall cycles) and insufficient space (thrashing).

In our thesis, we assume that memory space is available in sufficient quantity and that bandwidth is the limiting factor. With the growing size of the main memory found

in typical systems, this is a reasonable assumption. Contention for memory space is beyond of the scope of this thesis.

The situation is different for caches, which are typically small. On the one hand, utilization defined by access frequency is the right metric for temperature-aware scheduling, since more accesses to the cache mean higher power consumption and temperature. On the other hand, however, for avoiding contention, cache space is much more important than cache bandwidth: If a cache has multiple ports, accesses from different cores or logical processors can happen in parallel, and there is no contention when different portions of the cache are accessed. Rather, contention results when tasks whose working sets do not fit into the shared cache simultaneously run in parallel and evict each other's data from the cache.

For many applications, frequent cache accesses coincide with a big working set and a high probability of evicting other tasks' data, so there is a correlation between access frequency and contention. Yet, using the size of the working set in relation to the size of the cache or even a more sophisticated metric like the reuse distance profile [BH04] as a measure for representing cache behavior would be more accurate. However, information about the working set and the cache behavior cannot be obtained easily, but would require a computationally expensive model, special hardware support, instrumentation, or off-line tracing of applications [AHH89, BS96, SDR02, FSSN05, ZII+07].

Since we cannot easily obtain a more sophisticated metric on-line and with low overhead on today's hardware, we use access frequency to represent cache utilization. In Section 5.4, we will discuss in more detail why this is a viable approach for shared caches of CMP chips such as the Intel Core2.

Ultimately, the better solution would be to allow a scheduler to specify whether the activity monitor should provide utilization of space-constrained resources like memory or cache in terms of access frequency or space, or even represent both as separate vector components.

In the rest of this thesis, when we use the terms "memory utilization" or "cache utilization", we will be referring to bandwidth utilization of the respective resource unless stated otherwise.

3.3 Versatility and Portability

In the past, a number of scheduling policies have been proposed that use metrics related to the utilization of various on-chip resources to avoid thermal problems [GPV04, DM06, KSPJ06] and to determine what tasks to co-schedule on SMT or CMP processors [PELL00, NP02, BP05, MAN05, EMGAD06, ZDFS07].

While we discuss the these approaches in more detail in Chapters 4 and 5, we note here that all previous approaches have one common property: There is a close coupling between the metrics used to characterize tasks and the respective scheduling

policies, i.e., the metrics were specifically designed and selected for the respective policy, and the policies were designed to consider fixed, and often microarchitecture-specific metrics. To our knowledge, the general characterization by resource utilization we propose as a new abstraction to be used for diverse scheduling policies with varying goals is new.

We argue that it is possible to specify scheduling policies in a way that is independent from a particular microarchitecture or even architecture, if information about the implications of running a particular task is provided in a generic way. The utilization of chip-related resources is a metric that suggests itself for this purpose because of its close relation to thermal phenomena and to the interaction of execution contexts (logical processors, CMP cores) via shared resources. Yet, resource utilization is a generic concept that is common to all architectures and microarchitectures.

Activity vectors enable flexible scheduling policies by providing a layer of abstraction from the resources of a particular platform. As a simple example, an algorithm "do not co-schedule tasks that use the resource memory" can be transformed into an algorithm "do not co-schedule tasks that use the resource cache" simply by changing the component of the activity vector the algorithm considers; the algorithm need not even be aware of the actual resource for which it avoids contention.

On the same basis, it is possible to implement generic scheduling algorithms that can be applied even if it is unknown what are the most critical resources for the platform the algorithm is running on; such algorithms could be as simple as "avoid running tasks using the same resources together" for mitigating resource contention or "avoid running tasks using the same resources in succession" for mitigating hotspots. As we will show in Section 5, we were able to successfully reduce contention on a multicore chip of the Core2 microarchitecture and on a multithreaded chip of the NetBurst microarchitecture using exactly the same scheduling algorithm, with the only difference being the underlying implementation of activity vectors.

The concept of activity vectors decouples the mechanism of determining resource utilization, which is (micro)architecture-dependent, from the actual scheduling policies, which are often architecture-independent. For example, determining the utilization of the memory bus involves different counters on different architectures, but an algorithm for avoiding memory contention by co-scheduling tasks can be formulated independently from a particular architecture.

Hence, a vector-based scheduling algorithm can be used on any platform for which there is a suitable implementation of activity vectors. Vice versa, activity vectors need only be implemented once for a particular platform, and then different scheduling algorithms can build upon them.

At the same time, the concept of activity vectors still enables a great degree of flexibility and customizability, since the number of components and the actual resources considered can be adapted to the specific characteristics of each architecture. For instance, for an SMT processor with many resources shared between siblings, the ac-

tivity vector can have a large number of components and cover a large number of resources, whereas for a CMP processor with only cache and memory as shared resources, fewer components suffice. Another criterion for the number and the selection of resources represented by the activity vector are the facilities the particular platform offers for determining resource utilization, for example, the number of performance monitoring counters and the selectable performance events.

3.4 Activity Vector Framework

For applying vector-based scheduling in an operating system, we introduce an activity vector framework, the general architecture of which Figure 3.1 depicts. The framework consists of three components: The *activity monitor* for observing task characteristics, the *activity vectors* for representing the characteristics, and the *vector-based scheduler*, which performs scheduling guided by task characteristics.

We introduce the activity monitor as a new component into the operating system. The activity monitor observes the utilization of processor-related resources and uses this information to maintain the task activity vectors. Periodically, the activity monitor updates the activity vector of the currently running task with the resource utilization observed during the execution of the task. We represent activity vectors by extending the task control block (TCB), a data structure used by the operating system to represent task state. A vector-based scheduling policy can then take advantage of the characterization provided by activity vectors to make optimal use of the processor and its resources.

3.5 Determining Activity Vectors

For determining a task's activity vector, the activity monitor samples the utilization of resources during the execution of the task. In our case, we need to sample the utilization of processor-related resources, i.e., of the units found on the processor, and the utilization of the memory interconnect.

Information about utilization could be provided directly by the hardware, for example via special registers. Unfortunately, this is not the case in today's processors. Isci and Martonosi [IM03] has shown that for an Intel Pentium 4 processor, the degree of utilization for each chip unit can be determined using performance monitoring counters. The method suggested by Isci and Martonosi attributes events or combinations of events to each chip unit and determines the utilization of the units by counting how many events occur in a given timeframe. We adopt this methodology to determine activity vectors for the NetBurst microarchitecture of the Pentium 4 (Chapter 4), and use a similar methodology for determining activity vectors for the Intel Core2 microarchitecture (Chapter 5).

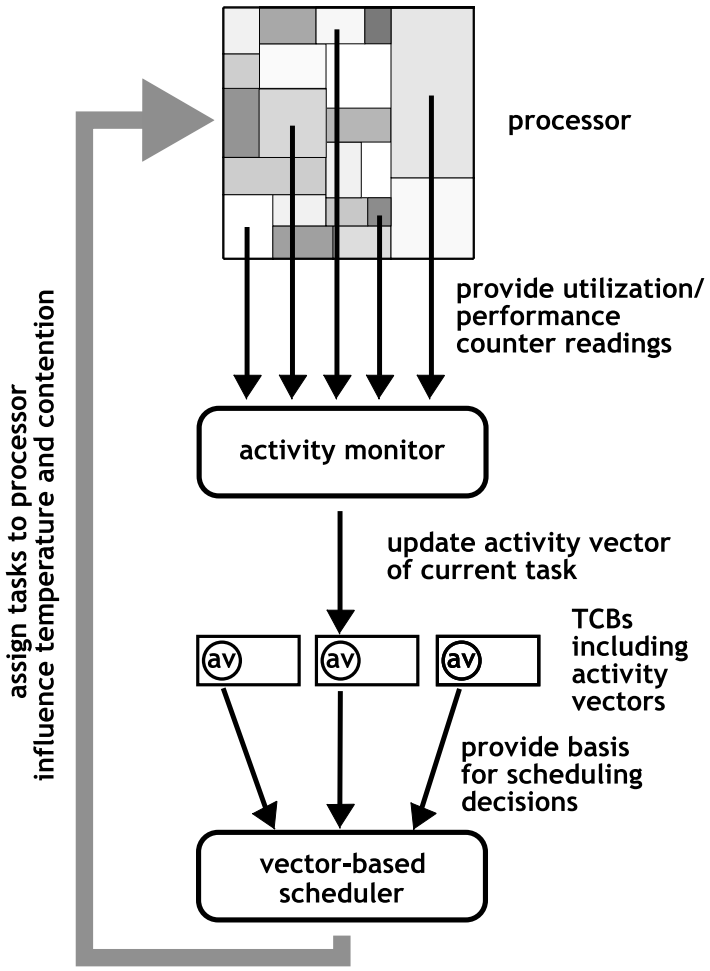


Figure 3.1: Structure of the activity vector framework

To characterize tasks, we need to attribute the utilization of chip resources to the task that caused them. Therefore, we need to sample utilization at least on every task switch. On some architectures, the limited number of performance monitoring counters necessitates multiplexing counters, i.e., switching between different counter configurations, for capturing all relevant events [IM03]. Therefore, we perform additional sampling at timer interrupt granularity. This way, we obtain event counts of all counter configurations during a task's timeslice.

Thus, on each timer interrupt and on every task switch, the activity monitor reads the necessary performance monitoring counters, compares the event counts to the saved values from the last sampling point, calculates the number of accesses to the individual resources, and, using the theoretical maximum number of accesses that could have occurred, determines the utilization of the resource. If necessary, we re-program the event monitoring counters to monitor different events for the next sampling period in order to accomplish multiplexing.

Multiplexing at constant intervals has the potential of introducing systematic errors if the tasks monitored change their behavior at the same frequency as the multiplexing interval. In addition, malicious programs could purposefully adopt a certain behavior during the intervals during which they are monitored in order to manipulate their activity vector. Anderson et al. [ABD+97] proposes to randomize the sampling interval length used for task characterization in order to avoid systematic errors; the same mechanism is suitable to counteract manipulations [ZDFS07]. Accordingly, our design could be adapted to multiplex event monitoring counters not at every timer interrupt, but after a (small) random number of timer interrupts.

In case of multicore or multithreaded processors, we need to address the problem that for a shared resource, multiple tasks running in parallel on the cores or logical processors can be responsible for the utilization of the resource. Therefore, the activity monitor needs to distinguish which core or logical processor (and thus which task) issued the instruction that has lead to an event that has occurred on the chip. For the microarchitectures we consider in this thesis (NetBurst and Core2), the performance monitoring facilities generally allow to configure the counters in a way that they only count events issued by a specific core or logical processor, thus allowing to attribute events to individual tasks. However, for the NetBurst microarchitecture, some events can only be counted globally, i.e., they can not be attributed to a logical processor (see Chapter 4.5.1). This introduces error into the corresponding components of the activity vectors. Yet, the evaluation of our scheduling policies shows that activity vectors on the NetBurst microarchitecture are accurate enough for the policies to yield significant benefit.

The activity vector of a task is not constant, but can change over time as the task passes through different phases, e.g., runs different algorithms successively. Therefore, the activity monitor continually updates the activity vector of each task that is

executed on a processor using the information about resource utilization sampled during the task run.

The optimal base for scheduling decisions would be the future characteristics of the tasks, especially the characteristics for the next timeslice. Since we determine activity vectors on-line, the future behavior of tasks is unknown. However, if a task shows stable behavior, i.e., the characteristics of the past persist for the near future, and phase changes only occur from time to time, past characteristics are a good proxy for future characteristics. This is the case for many applications: An analysis of the SPEC CPU 2000 benchmarks by Isci et al. [IMB05] revealed that of 25 benchmarks, 17 spend more than 70% of their runtime in phases longer than 200ms (which is considerably longer than scheduling time quanta).

On the other hand, even within a relatively stable phase, the utilization of chip resources is not fixed, but varies to a certain degree, though not as strongly as between different phases. We consider variations that occur on a timescale shorter than a timeslice as noise, since those variations are irrelevant to scheduling; especially we should not make scheduling decisions based on short-term changes in task behavior, since the consequence of a scheduling decision (the dispatching of a task to a processor), lasts for an entire timeslice.

We performed an analysis of the SPEC CPU 2006 benchmarks by sampling their unit utilization over a complete run at millisecond granularity. As an example, Figure 3.2 shows the phase behavior of the SPEC benchmark `astar` (which performs A* path finding algorithms [HNR68]). The figure depicts the utilization of the memory bus, the L2 cache and of the chip components not shared between cores (accumulated) as sampled on an Intel Core2 processor. On the one hand, the figure clearly shows long phases that last up to minutes. These phases stem partly from the fact that the `astar` benchmark consists of three distinct algorithms [Hen06], but the total number of phases being larger than three indicates that there are also phase changes within the individual algorithms. On the other hand, the figure also depicts short-term variation of utilization within phases (the noise), which manifests itself in the vertical breadth of the depicted curves.

The example of `astar` unites both extremes that we could also observe with the other SPEC benchmarks: on the one hand, long, differentiated phases, and, on the other hand, short-term changes that we need to consider as noise. For providing meaningful activity vectors, we need to address both cases. We need to adapt a task's activity vector whenever there is a change between long-term phases, but we also need to filter out the noise in order to avoid fluctuations of the activity vector that could provide the scheduler with inaccurate information.

As a solution, we calculate each component of the activity vector as an exponential moving average over the utilization of the respective chip unit. The exponential average is a weighted moving average that weights sample values with an exponentially lower weight the further they lie in the past.

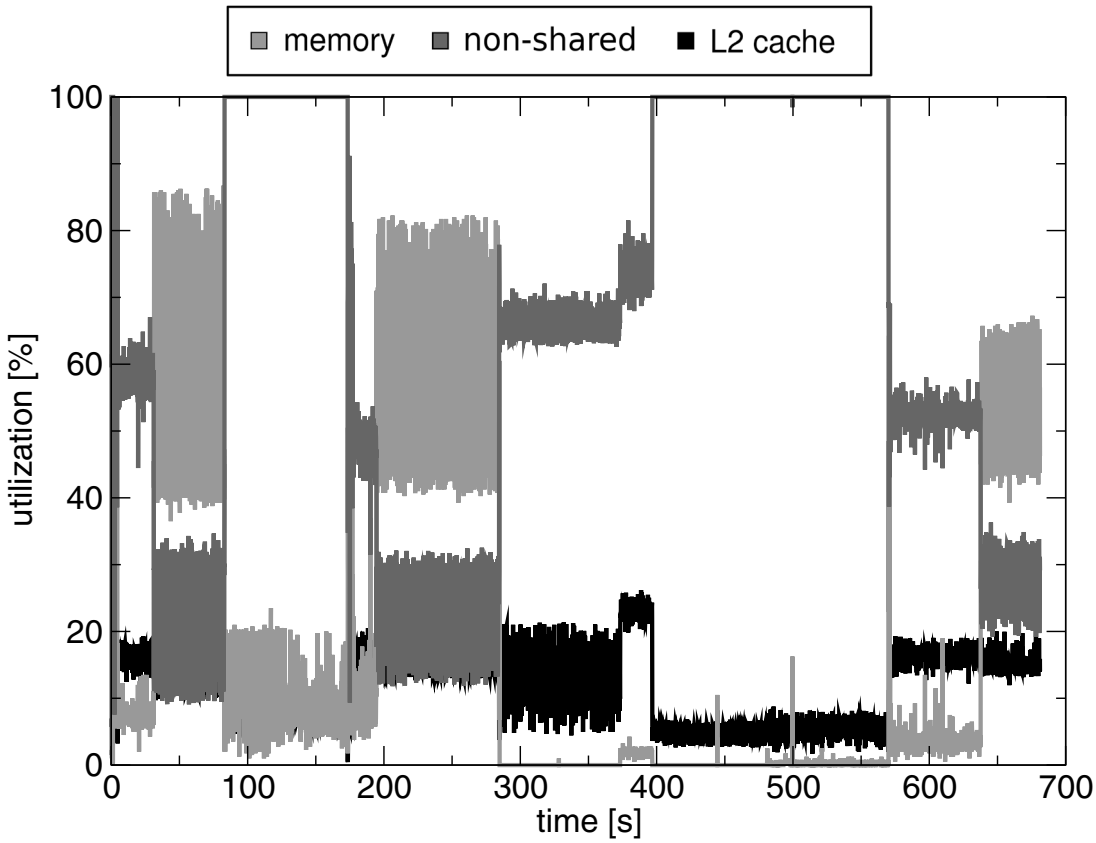


Figure 3.2: Phases of the astar benchmark

As a result of averaging, short term changes in a task's behavior do not cause the task's activity vector to change significantly, whereas—owing to the exponential fading of the past values' contributions—a permanent change is reflected in the activity vector after an appropriate time (the order of a timeslice, as can be adjusted by selecting appropriate weights for the moving average). Thus, the exponential average acts as a low-pass filter and smoothes out changes in a task's behavior that are only of short duration. This way, we avoid erroneously changing the task's activity vector if its characteristics show temporary fluctuations.

3.6 Implementation

We implemented activity vectors for the Linux kernel and for two microarchitectures, the NetBurst microarchitecture and the Core2 microarchitecture.

As mentioned in Section 3.4, we represent task activity vectors as part of the task control block (TCB), the data structure the operating system uses for representing task state. Thus, for implementing activity vectors in Linux, we supplement the `task_struct` data structure, which implements the TCB in Linux, with an array holding the components of the task activity vector.

As mentioned, the range of an activity vector's components lies between 0 and 1. In Linux, however, kernel code is not supposed to use floating point arithmetic in order to avoid saving and restoring the contents of the floating point registers upon kernel entry or exit. Thus, we implement activity vectors using fixed-point arithmetic and scale the vector components by a constant factor to transform them to an integer range.

We implement the activity monitor as a microarchitecture-specific routine that we invoke from the timer interrupt handler, as well as from the scheduler prior to a task switch. The activity monitor updates the elements of the array representing the activity vector in the TCB of the task that is currently running (invocation from the timer interrupt handler) or that is about to lose control of the processor (invocation from the scheduler prior to a task switch) by calculating the exponential moving average over the utilization of processor resources as described in the preceding section. Calculating the exponential moving average does not require a history of sampling values, but merely the current sample value and the exponential average of the last sampling period (i.e., the old value of the activity vector's respective component).

For determining unit utilization on the NetBurst microarchitecture, we implement the methodology proposed by Isci and Martonosi [IM03]. For the Core2 microarchitecture, we use a similar methodology, which is detailed in Chapter 5.

In addition to providing activity vectors to the scheduler, we also export the information provided by activity vectors to userspace via Linux's `/proc/<PID>/` interface for informational purposes.

3.7 Overhead

Activity vectors provide information to be used by schedulers. One main goal of many scheduling policies is achieving optimal performance. Hence, it is a primary requirement for activity vectors to cause low overhead. In particular, for performance-oriented policies, the overhead required for acquiring activity vector information needs to be outweighed by the performance improvement achievable with the vector-based scheduling policy.

Since vector-based scheduling by definition relies on the characteristics of the tasks present in the system, and since we do not want to make any assumption on the workload, it is possible that a vector-based scheduling policy does not yield any performance improvement for a given situation. Especially if all tasks present in the system have similar or equal characteristics, vector-based scheduling cannot lead to better decisions than conventional scheduling policies. If determining activity vectors had a significant overhead, vector-based scheduling would in total yield worse performance than traditional scheduling for those situations. Thus, we demand that activity vectors involve *close to zero* overhead.

Since sampling utilization for updating activity vectors only happens periodically, this is a feasible goal if the time required for updating the vectors is negligible in comparison to the sampling interval length. Reading a performance monitoring counter typically takes in the order of 10 to 100 cycles (e.g., we measured 140 cycles on an Intel Pentium 4 or 54 cycles on a Core2, using the `rdpmc` instruction), whereas with Gigahertz clocks and timer interrupts in the range of milliseconds, a sampling interval encompasses millions of cycles. Calculating utilization from the counter values involves only few arithmetic operations.

While generally, the time required for reading one counter is low compared to the length of the sampling interval, the overhead can accumulate if a large number of counters need to be read, or if multiplexing requires additional writes to configuration registers, which are often more costly than reads (e.g., 950 cycles for writing a configuration register on a Pentium 4). As an example, sampling unit utilization for the NetBurst microarchitecture, which involves a total of 15 counters and multiplexing in our implementation, causes a total overhead of 33,000 cycles (see Section 4.6).

Our experiments with SPEC workloads presented in Chapter 4 indicate that activity vectors introduce an overhead in the percent range for the NetBurst architecture. For the Core2 architecture, where determining activity vectors only involves four counters and no multiplexing (Section 5), our experiments do not reveal any measurable overhead.

3.8 Vector-Based Scheduling

A *vector-based scheduling policy* is a scheduling policy that uses the information provided by task activity vectors in order to attain a specific goal, for example, increased performance, a better temperature distribution, or higher energy efficiency. In the following chapters, we will develop such scheduling policies.

The focus of our proposed policies lies on non-interactive, CPU-intensive tasks that do little I/O. In addition, our policies are intended for situations in which all execution contexts are utilized. For situations with idle contexts, there are straightforward policies for mitigating hotspots (e.g., leaving the hottest processors idle) or reducing resource contention (e.g., running the most resource intensive tasks on processors with idle siblings).

While we will detail the specific vector-based scheduling policies we propose in the following two chapters, at this point, we want to give an overview about the basic mechanisms of vector-based scheduling. Conventional scheduling policies used in today's operating systems leave several degrees of freedom unexplored. Vector-based scheduling seeks to exploit these degrees of freedom:

Scheduling order With round-robin–like scheduling policies, all tasks assigned to a processor are scheduled in turn, but the actual order in which tasks are scheduled is arbitrary. We propose *sorted scheduling*, which arranges the runqueues of the processors in a certain order that is defined by the tasks' activity vectors. An example of sorted scheduling is executing tasks that use different functional units successively in order to avoid having constantly utilized units that could become hotspots.

Co-scheduling As mentioned in Chapter 2, co-scheduling is not applied in today's general purpose operating systems. As a consequence, the combination of tasks running simultaneously on a chip is arbitrary. Vector-based co-scheduling controls the combination of tasks running at a time by scheduling tasks with certain characteristics together. An example for co-scheduling is to run tasks utilizing different functional units of an SMT processor in parallel to avoid contention.

Assignment of tasks to processors As described in the preceding Chapter, affinity-based SMP schedulers do load balancing to have an equal number of tasks running on each processor. Which tasks are migrated to countervail load imbalances is largely arbitrary. In addition, the way newly started tasks are distributed to processors in the first place usually follows no defined scheme other than not creating load imbalances.

Vector-based task placement and migration policies control the distribution of tasks to processors with the aim of creating runqueues with specific properties. An example for vector-based task assignment is to distribute tasks in a way that each processor is assigned tasks utilizing different functional units.

In many cases, proper task-to-processor assignment is a prerequisite for effective sorted scheduling and co-scheduling, since it determines the contents of the individual runqueues, which in turn determines the available options for sorting and co-scheduling.

3.9 Uniform vs. Non-uniform Policies

We define two ways a scheduling policy can use activity vectors: the uniform and the non-uniform view. A scheduling policy that has a *uniform* view on activity vectors (*uniform policy*, for short) does not associate a particular meaning with the individual vector components other than the fact that each component represents the utilization of a resource. Which component stands for which particular resource is unknown to a uniform scheduling policy. A uniform policy thus threats all components equal. An example for a uniform policy could be "co-schedule tasks that use mutually different resources".

A scheduling policy that has a *non-uniform* view on activity vectors (*non uniform policy*) is aware of additional properties of the resources that the vector components represent and thus associates a meaning with the individual components. A non-uniform policy could be "co-schedule tasks utilizing resource x either with other tasks utilizing resource x or with tasks utilizing resource y, but do not co-schedule two tasks using resource y" (resource x could be the integer unit and resource y the memory bus of a multicore system). Since activity vectors abstract from the actual chip resources, even a non-uniform scheduling policy need not know what resources x and y actually represent, but it needs to be aware that two tasks that both utilize resource y interfere with each other, whereas two tasks that both utilize x or one task that utilizes x and one that utilizes y do not interfere. Thus, a non-uniform policy needs to distinguish between vector components and associate properties with individual components that are not applicable for other components.

While uniform policies are more generic and can be designed with less knowledge about the properties of the hardware, non-uniform policies can consider the characteristics of a particular platform. For example, the fact that two compute-bound tasks running on two different cores of a multicore chip do not influence each other, whereas two memory-bound tasks do, can be expressed using a non-uniform view on activity vectors, but not using a uniform view. Thus, the non-uniform policy in the example above can be expected to yield better performance on a CMP than the more generic uniform policy mentioned at the beginning of this section, since it allows to consider the characteristics of the given architecture. On the other hand, the non-uniform policy is beneficial if applied to different cores, but not if applied to SMT siblings, since in an SMT processor, there is interference between tasks that utilize the integer unit. The uniform policy that simply avoids co-scheduling tasks that utilize the same resources is beneficial on both platforms.

In general, we can say that uniform policies have a broader scope, whereas non-uniform policies allow optimizations for a particular range of platforms, such as CMP or SMT.

4 Temperature-Aware Scheduling

4.1 Introduction

Increasing clock rates in combination with rising integration densities have led to an aggravation of thermal problems in recent generations of microprocessors. Owing to dramatically increasing cooling costs, processor packaging and cooling is no longer designed for the worst case, but rather for more moderate demands that realistic applications are assumed not to exceed (Thermal Design Power). As mentioned in Chapter 2, strategies for dynamic thermal management are applied to avoid overheating the processor in case of violation of the thermal design point.

In general, the temperature of a chip cannot be described accurately by a single temperature value. Different units on the chip such as ALUs, FPUs, or caches have different structures and thus different power densities, resulting in a non-uniform temperature distribution within the chip. In addition, the actual temperature distribution within a chip depends on the switching activity in the individual chip units, which is determined by the instruction mix running on the processor: Chip units that are not needed to process a certain instruction are inactive and have a lower power consumption than when active.

Moreover, mechanisms for clock and power gating have been introduced to limit the overall power consumption of the chip. Clock gating deactivates the clock signal for units that are currently not in use; power gating completely deactivates the power supply for those units. Despite the positive effect these mechanisms have on overall power consumption, they are not suitable for reducing localized heating in permanently active areas of the chip, but instead increase thermal imbalances between active and inactive areas.

Different programs differ in the type of instructions they execute, depending on their functionality, on the way they are written, and on the compiler that was used for translating them. There are programs that issue many integer instructions, but no floating point instructions, or floating point programs that contain only few integer instructions, programs that do many memory accesses and programs that almost only work on registers. Thus, the program that the processor currently executes influences the distribution of power density on the processor die and for that reason also the distribution of temperature. We observed variations of up to 30 Kelvin for the temperature of chip units between different applications from the SPEC CPU2006 benchmark suite on an Intel Pentium 4 Xeon processor.

Preventing thermal emergencies requires throttling the processor as soon as the temperature of the hottest part of the chip surpasses the critical temperature limit, no matter if there are other units whose temperature is far below the limit. Throttling reduces the processor's power consumption, but also its performance, and should therefore be avoided.

In this chapter, we investigate to what extent the scheduler can influence the temperature distribution within a processor. As mentioned before, the temperature distribution of a chip is determined by the software that runs on the chip. If, for example, the integer ALUs of a chip are near the critical temperature, and an integer task is scheduled next, the processor is likely to overheat soon, which results in throttling. On the other hand, if a floating point task is scheduled next, the integer ALUs remain mostly idle and can cool down. Thus, balancing temperature within a chip—which means preferably scheduling tasks using cool units—avoids throttling.

The decisions of the operating system's scheduler influence the need for throttling a processor, which in turn influences the system's performance. We argue that in order to achieve maximum performance, the scheduler needs to know about the characteristics of each task, that is, which units of the chip a particular task uses, and to what degree it utilizes these units.

Based on task activity vectors delivering information about the utilization of individual chip units, we propose and evaluate the following scheduling strategies [MB08b]:

- scheduling tasks that use different units successively
- distributing tasks among processors of a multiprocessor system so that each processor executes tasks that use different units
- running tasks that use different units simultaneously on multithreaded processors.

Our approach of temperature-aware scheduling is best-effort, meaning that our policies strive to improve temperature distribution, but cannot guarantee to avoid overheating. In particular, the success of our policies depends on the workload. For instance, if only tasks are available that all cause a hotspot at the same location, our policies cannot avoid the formation of such a hotspot. Saving the hardware from overheating is the responsibility of hardware mechanisms like the Catastrophic Shutdown Detector (Thermal Monitor 1 and 2) used in the Intel processors [Int06], which avoid overheating at the cost of introducing performance penalties. Though not being able to avoid overheating in all cases, our policies reduce the need for engaging thermal throttling by improving temperature distribution.

We implemented the scheduling strategies mentioned above for the Linux kernel. As our evaluation platform, we chose the Intel Pentium 4 Xeon processor. The reason for this choice is that the NetBurst architecture of the Pentium 4 features a rich set of

performance monitoring counters that allow determining utilization at the granularity of individual chip units, and that simultaneous multithreading is supported, so we can study the implications of this feature on temperature distribution. In addition, the Pentium 4 is known as a thermally challenging processor.

We verified the benefits of our policies with experiments using an eight-way multiprocessor system. We perform task characterization and scheduling on the real physical system, and, for observing temperature distribution on the chip, resort to the HotSpot [HSS+04] temperature simulator, which we feed with power samples acquired during the test run. Our experiments show that vector-based, temperature-aware scheduling achieves a more balanced temperature distribution than standard Linux scheduling and reduces hotspots.

The rest of this chapter is structured as follows: Section 4.2 explains how we obtain information about the temperature distribution on a processor chip. Section 4.3 discusses related work in the area of temperature-aware scheduling. Section 4.4 presents the design of our vector-based scheduling policies. Section 4.5 outlines the implementation for Linux. Section 4.6 evaluates our design. Finally, Section 4.7 concludes the chapter.

4.2 Determining Chip Temperature

The scheduling policies we propose in this chapter aim at improving the temperature distribution on the processor chip. To verify the benefits of our policies, we need a way of observing the temperature distribution. Moreover, some of our proposed policies need the temperature distribution on the chip in addition to the tasks' activity vectors as a basis for deciding which task to schedule.

Traditionally, strategies for avoiding overheating a processor are based on a single *chip temperature* value [GBCH01, DM05, MB06]. This value usually stems from a thermal diode located somewhere on the processor [Int02], or is assembled out of several temperature readings, if multiple sensors are present on the chip [NRM+06, RHAH06].

However, physical sensors have several limitations: Firstly, their number is limited, since increasing the number of sensors is expensive in terms of die area. Hence not all possible hotspots can be covered. Secondly, thermal sensors often cannot be placed directly at the hottest chip units for topological reasons. Thirdly, when analog temperature sensors (thermal diodes) are used to read temperature via the super-I/O chip and the system management bus, reading temperature takes a long time and cannot be performed more than several times per second [GBCH01, HKK06]. (This is no longer a concern for temperature sensors whose readings are digitized on-chip and made accessible to the software via model-specific registers [RHAH06, NRM+06].)

The consequence of the limitations of physical sensors is that either some parts of the chip can possibly overheat without the sensors detecting it, or that the critical

temperature at which response mechanisms in software or hardware are engaged must be set to a lower value to compensate for the sensors reporting values that are lower than the maximum temperature somewhere on the chip. The latter, however, may lead to engaging thermal management without need and thus needlessly sacrificing performance.

The temperature monitoring facilities that today's commodity processors offer are not suitable for accurately observing the temperature distribution on the chip. We experimented with several Intel Pentium 4 and Core2 processors by starting thermally demanding tasks on a formerly idle processor. The temperature values reported by the sensors of the Pentium 4 chips only changed slowly over the course of several minutes, which indicates that the values the temperature monitoring facilities report are more representative of the heat sink temperature than of the silicon temperature. For the Core2, a more immediate temperature change was visible, which suggests that the digital temperature sensors deployed in these chips are indeed representing silicon temperatures. However, the Core2 only offers one temperature reading per core, which does not allow to observe temperature distribution within the cores.

Bellosa et al. [BWWK03] proposes to use performance monitoring counters in combination with an energy model and a thermal model for estimating the power consumption and temperature of a processor chip as a whole. Lee and Skadron [LS05] uses a more complex thermal model in combination with energy estimation by performance monitoring counters to obtain the temperature of individual parts of the chip: As explained in Chapter 3, the information obtained from performance monitoring counters allows to estimate the degree of utilization of individual chip units. The power consumption of each unit, in turn, depends on the unit's utilization, and can thus be estimated when utilization is known. Using a thermal model, temperature estimation for the blocks of the chip based on their power consumption is possible [LS05, HKK06].

Estimation of power consumption from performance counters yields an error of less than 10% [BWWK03, IM03]. The thermal model proposed by Lee and Skadron has been verified against a test chip and yields a worst case error of 7% for temperature estimation [HSS+04]. Hence, using estimation and simulation allows to obtain realistic temperature values with finer granularity than achievable with hardware sensors, not only in terms of space (obtaining temperatures for each unit is possible), but also in terms of time. The latter is determined by the rate at which the performance monitoring counters are sampled, which can be done at every timer interrupt, that is up to 1000 times a second in today's systems.

The methodology just described enables us to verify the benefits of our proposed scheduling policies, since it allows us to observe the course of temperature for each chip unit at a high enough temporal and spacial resolution, which is not possible using the sensors provided by the hardware.

4.3 Related Work

In the past, a number of approaches have leveraged the characteristics of individual tasks for improving temperature distribution. Related work that has addressed thermal problems by using task characterization can be categorized into software approaches that are guided directly by temperature, software approaches guided by utilization, and approaches at the hardware level using special hardware features. A further approach that consists in using shorter timeslices does not directly apply task characterization, but relies on workload diversity being present.

4.3.1 Approaches guided by temperature

Some previous approaches that have addressed thermal imbalances within a chip by means of scheduling have characterized tasks by the temperature they cause when running on the chip.

Kursun et al. [KCBB06] investigates the impact of the order in which tasks are scheduled on the temperature distribution within the chip. The authors find that temperature distribution can be improved considerably by scheduling, which mitigates the need for thermal throttling. They propose a scheduling policy that is similar to our approach of temperature-aware scheduling and selects tasks that are expected not to stress the parts of the chip that are currently hot. To provide input to the scheduler, tasks are profiled by reading on-chip temperature sensors. The authors circumvent the problem that today's hardware often does not provide the necessary number of sensors for doing detailed task characterization by using a simulator for their studies instead of real hardware. Apart from using temperature for characterization, the difference to our approach is that only single-processor systems are considered, and that Kursun et al. does not address the problem of implementing the proposed policy in an operating system.

Choi et al. [CCF+07] evaluates the potential of mitigating thermal problems by operating system scheduling and comes to the conclusion that scheduling offers the potential of mitigating hotspots with negligible overhead. The approach uses temperature sensors to categorize tasks into "hot tasks" and "cold tasks", which is a more coarse-grained characterization than we use in our work and does not consider the location at which heat is dissipated. Similar to our approach, the authors use a modified load balancing algorithm in order to balance the number of hot and cold tasks on the processors. In addition, they defer the execution of hot tasks in favor of cool tasks if high temperatures are detected .

Coskun et al. [CRW07] addresses temperature-aware scheduling for multicore system-on-chips. The work distributes tasks to cores in a temperature-aware fashion using a probabilistic load balancing algorithm that sends tasks to cooler cores with higher probability than to hotter cores. Similarly, Stavrou and Trancoso [ST07] proposes scheduling algorithms that place tasks on cores of a CMP that have a low

temperature or have neighboring cores with low temperatures. In contrast to our approach, both do not address temperature distribution of functional blocks within a processor core, and do not directly characterize tasks by temperature, but take advantage of the fact that if a task is assigned to a core, this core will have a higher temperature than an idle core.

Using temperature for characterizing tasks has several drawbacks. Since today's processors typically feature only a small number of temperature sensors, temperature distribution can only be inferred at a relatively coarse spacial granularity, and it is not possible to characterize tasks at the level of chip units to address temperature distribution within a core. Also, using temperature values is inadequate if tasks running on different chips are to be compared, since the cooling situation of the chips may differ. For instance, one chip can be located more closely to a fan or air inlet than another. Therefore, a task that is found to result in a certain temperature on a certain chip need not necessarily lead to the same temperature if migrated to another chip. Temperature is also not suitable as a metric if tasks are running on SMT siblings in parallel, since in that case, the resulting temperature is a mix of the characteristics of the tasks running on the siblings.

An alternative approach that solves some, but not all of these problems, is characterizing tasks by the power consumption they cause instead of by temperature. The concept of *task energy profiles* we proposed in an earlier work [Mer05] allows an energy-aware scheduler to balance the temperature of physically different chips, but not to address temperature distribution within a chip. Unlike a characterization by utilization of processor resources, the energy profile does not provide information about where on the chip the power is dissipated.

4.3.2 Approaches guided by utilization

We are aware of two previous approaches at mitigating thermal problems that have used a characterization of tasks by utilization of chip units.

Gomaa et al. [GPV04] proposes combining tasks with different characteristics on SMT siblings to avoid overheating individual chip units while other units are cold. The authors propose to characterize tasks by instructions per cycle (IPC), the utilization of integer and floating point resources, or the utilization of chip resources prone to overheat, and propose several policies for creating heterogeneous workloads. Gomaa et al. states that task characterization and temperature-aware scheduling are supposed to be done at the operating system level, but uses simulation for evaluation instead of real hardware and a real operating system. The paper does not discuss how exactly the characterization by resource utilization should be done, and how operating system scheduling can put the proposed policies into practice.

Donald and Martonosi [DM06] studies the application of DVFS in combination with task migrations to prevent hotspots in multicore processors. The approach uses performance monitoring counters to assess tasks in respect of the thermal intensity

they show for certain chip resources such as integer and floating point units and register files. When a chip unit reaches a critical temperature, a migration to another core is initiated. The approach is comparable to our migration policy, which also strives at avoiding hotspots by distributing tasks to CPUs accordingly. The difference between the approaches is that Donald and Martonosi only considers scenarios with exactly one task per core and rotates tasks to ameliorate temperature distribution. Our approach, in contrast, considers scenarios with multiple tasks per processor core, uses migrations to provide tasks with differing characteristics for each processor, and, in addition, modifies the order in which tasks are scheduled on a processor to attain a better temperature distribution.

A third approach, Kumar et al. [KSPJ06], uses performance monitoring counters to determine the utilization of functional units on the chip, but aggregates the utilization of all functional units into a single temperature estimate that is then used to characterize tasks. Therefore, the approach only addresses overall chip temperature but cannot leverage the fact that different tasks can dissipate heat at different locations.

4.3.3 Approaches at the hardware level

Many approaches that aim at mitigating thermal problems by leveraging workload characteristics address those problems at the hardware level rather than in the operating system. In contrast to the operating system, the hardware has no knowledge about tasks, so in order to exploit the characteristics of different threads of execution, policies in hardware operate at the level of hardware threads, e.g., the logical processors of a multithreaded chip.

Donald and Martonosi [DM05], like our approach, exploits the characteristics of individual threads of execution to mitigate thermal problems. Donald and Martonosi uses an adaptive fetch policy or register renaming policy for multithreaded chips to avoid the formation of hotspots. The approach uses performance monitoring counters to determine the utilization of the integer and the floating point unit for each thread. Depending on the temperatures of these units, the processor prefers threads using cooler units by fetching more instructions for these threads than for threads using hotter units. In contrast to our approach, reducing hotspots is accomplished at the expense of fairness. Moreover, an adaptive fetch policy is only beneficial if tasks showing different unit utilizations are co-scheduled on the logical processors. Since the hardware cannot influence the mapping of tasks to hardware threads, a hardware level approach such as an adaptive fetch policy could benefit from operating system policies like the ones we propose, ensuring that tasks with different characteristics are co-scheduled.

Winter and Albonesi [WA08] addresses thermal non-uniformity in clustered SMT architectures. Based on a characterization of logical threads by temperature or activity of the floating point and integer units, the approach leverages the physically distinct backends of the clustered SMT architecture and assigns threads that caused a hotspot

in a particular backend to another backend that exhibits a low temperature for the resources the thread utilizes most. Winter and Albonesi shows that such a hardware-based steering mechanism reduces the need for throttling in order to avoid thermal emergencies.

Another hardware related approach to avoiding hotspots consists in adding spare resources such as register files, issue queues, or ALUs on the processor chip, and to migrate computation to one of those spare resources when the original resource reaches a critical temperature during the execution of a thermally challenging workload [HBA03, SSH+03]. This requires additional chip units, which occupy additional die area.

Mechanisms like adapting the fetch policy or migration to spare chip units are not available to the software, since the hardware typically does not expose these mechanisms. On the other hand, some mechanisms relevant to thermal behavior, like the migration of tasks between chips or the switching between tasks on a processor are not available to the hardware. In our opinion, optimal results can only be achieved if hardware and operating system software cooperate. One solution would be to allow the software to control hardware mechanisms. An example for this is the approach of Cazorla et al. [CKS+04], which proposes a modification to SMT hardware that allows the operating system to set priorities that the hardware considers when processing instructions of different logical processors. A similar mechanism is present in the IBM Power5 processor [KST04]. In the remainder of our work, we will not consider such hardware features, since they are not available in the majority of today's processors.

4.3.4 Shortening timeslices

Michaud and Sazeides [MS06] proposes to use shorter timeslices to prevent overheating individual parts of the chip, based on the rationale that with shorter timeslices, tasks using different resources are switched in quicker succession, so hotspots cannot form. We argue that shorter timeslices are only beneficial if there are tasks available that use different chip resources, and that the order in which they are scheduled is important. Our work addresses both problems. Our evaluations show that the approaches of shortening timeslices and vector-based scheduling can be combined to attain maximum benefits.

4.4 Vector-Based, Temperature-Aware Scheduling

As mentioned at the beginning of this chapter, the scheduling decisions of the operating system—determining what task the processor shall execute next—also determine the temperature distribution on the chip. Up to now, scheduling strategies found in general purpose operating systems like Linux or Windows are oblivious to this fact.

Today's schedulers make their decisions according to criteria like task priorities, fairness or good interactive performance, but neglect the impact that the order in which tasks are executed has on temperature distribution.

At the same time, in many scheduling strategies the exact order in which tasks are executed is unspecified. For example, in round-robin scheduling, it is not relevant in which order the tasks are scheduled, as long as each task gets its timeslice in turn and all tasks make progress. The same holds true for proportional share scheduling. Even with priority-based scheduling, the order in which tasks having the same priority are scheduled is not specified.

This can be used to influence the temperature distribution of the chip without breaking the properties and objectives of the respective scheduling policy. In the remainder of this section, we will concentrate on the widely-used round-robin policy.

Enabling the scheduler to influence temperature in a sensible way requires that we provide it with information about the characteristics of the tasks it manages, that is, what task utilizes which parts of the processor. The abstraction of activity vectors introduced in Chapter 3 provides this information. Task activity vectors characterize tasks by their utilization of processor resources. The activity of those resources is responsible for chip temperature and determines the temperature distribution. In the following, we propose scheduling strategies that use activity vectors to achieve a more balanced temperature distribution.

4.4.1 Runqueue sorting

We propose *runqueue sorting* as a scheduling policy that makes use of task activity vectors to avoid the formation of hotspots. We accomplish this by choosing a task to run next that does not utilize the parts of the processor that are currently hot, provided that such a task is available.

Our policy is based on the assumption that there are CPU-local runqueues, i.e., there is a list of tasks assigned to each CPU, and that the tasks in this queue are scheduled in round-robin fashion, that is, executed for a timeslice and then re-appended to the end of the queue thereafter. This holds true in many of today's operating systems, although the scheme is often slightly altered or enhanced. For instance, round-robin scheduling is often combined with a priority scheme, or the runqueue is split into several subqueues as is the case with the O(1)-Scheduler used in the 2.6 series of Linux up to version 2.6.22 [Jon06].

We propose two forms of runqueue sorting, a simple form and an enhanced form that uses a thermal model. The simple form arranges the tasks in the runqueue in a way that two successive tasks use resources of the CPU that are as complementary as possible, so the resources the previous task has used can cool down during the execution of the following task. The enhanced form considers the estimated temperatures of the CPU's units for choosing the next task in addition to activity vectors.

The simple form of runqueue sorting has a uniform view on activity vectors, since it does not associate a special meaning with the individual vector components and treats all components as equal. The enhanced form uses a non-uniform view, since it associates the vector components with particular chip units to be able to consider the temperature of these units.

Sorting efficiently

The set of tasks that compose a CPU's runqueue is not fixed. Tasks can terminate or block and thus be removed from the queue, and new tasks may be started. In addition, the load balancer can move tasks between the runqueues of different CPUs. Thus, keeping the runqueues sorted can cause a lot of overhead.

The alternative to sorting is searching the runqueue for a suitable task every time a scheduling decision is made. Searching, however is not scalable, since the time required to search for a suitable task grows with the length of the runqueue. Since scheduling tasks is a frequently invoked operation, scheduling algorithms have to be efficient. For example, Linux's O(1)-Scheduler scheduler is designed to satisfy an O(1) property, meaning that the cost of scheduling is independent of the number of tasks in the system.

We use a combination of searching and sorting to keep the overhead low. We adopt the scheme used in the O(1)-Scheduler of having two runqueues per processor, the first queue consisting of tasks waiting to be scheduled, and the second queue consisting of tasks that have been executed lately. When choosing the next task to be scheduled, we look at the first c tasks in the first queue and choose the best suited task out of them. After executing it for one timeslice, we append the task to the second queue. This way, the second queue is automatically sorted in a way that tasks using complementary resources follow each other. When the first queue is empty, we switch the queues. Since we operate on a queue now that has previously been sorted, there is a high probability that there is a suitable task among the first c tasks in the queue.

In addition, having two queues ensures that all tasks are making progress: A task that has been scheduled and is appended to the second queue is not scheduled again until all other tasks still residing in the first queue have been scheduled.

Example:

We consider a scenario with two types of tasks. Tasks of type A use different resources than tasks of type B. We assume there are five tasks of type A and three tasks of type B, we have chosen $c = 3$, and the initial order of the runqueue is $BBBAAAAA$. Figure 4.1 shows how the runqueues evolve under runqueue sorting. We underline the tasks the scheduler considers for making its decision, and mark the task the scheduler selects with a dot. Note that after sorting, there is still a succession of three tasks of type A since there are more tasks of this type than of type B. One might argue that it would

queue 1	queue 2
$BBBAA\underline{AA\dot{A}}$	
$BBBA\underline{AA\dot{A}}$	A
$BBB\underline{AA\dot{A}}$	AA
$BB\underline{\dot{B}AA}$	AAA
$B\underline{BA\dot{A}}$	$BAAA$
$\underline{B\dot{B}A}$	$ABAAA$
$\underline{B\dot{A}}$	$BABAAA$
$\underline{\dot{B}}$	$ABABAAA$
	$BABABAAA$
$BABAB\underline{AA\dot{A}}$	
...	

Figure 4.1: Example for runqueue sorting

be better to sort the queue in a fashion that distributes the tasks of type A more evenly, like $ABAABAAB$, but this is not feasible without global knowledge of the whole queue. In addition, the chip's units have typically already reached their peak temperature after one timeslice, so it does not matter whether we execute the supernumerous tasks of type A in groups of two or three.

Simple runqueue sorting

For the simple form of runqueue sorting, we use only the tasks' activity vectors for deciding which tasks out of the first c tasks in the queue we select for execution. If the timeslices are long enough (10ms to 100ms), which is the case in most operating systems, and a task switch occurs because of timeslice expiration, the unit utilization of the previously running task is a good indicator for the units' temperatures: Because of the comparatively small thermal capacitance of the chip units, the units the task has utilized are hot and the units the task has not utilized are cold. Therefore, we choose a task that uses different units than the one that ran before.

Whether a task uses different units than another task can be inferred from the angle the tasks' activity vectors form, as we want to explicate in the following. We denote the angle between two vectors $\vec{x}$ and $\vec{y}$ by $\angle(\vec{x},\vec{y})$.

Let $\vec{a}$ be the activity vector of the task that has run on the CPU up to now and $\vec{b_i}$ with $i \in \{1,..,c\}$ the activity vectors of the tasks we consider when choosing the next task to run. We want to choose the task whose unit utilization is most different from the unit utilization of the previously running task. In the ideal case, when two tasks use completely distinct units, their activity vectors are orthogonal, so $\angle(\vec{a},\vec{b_i}) = 90°$.

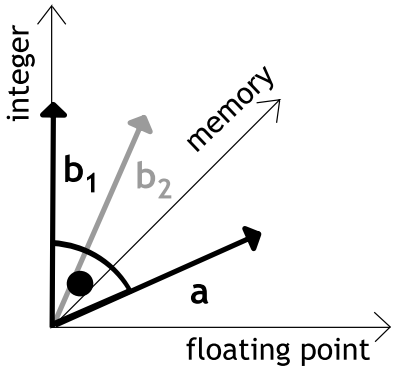


Figure 4.2: Angles between activity vectors

Hence, we select a task whose activity vector forms an angle with $\vec{a}$ as close as possible to 90°.

Figure 4.2 shows a simple example: Assume the task with activity vector $\vec{a}$ has utilized both the floating point units and the memory units, but not the integer units. In this case, the activity vector $\vec{b_1}$ of a task that only utilizes the integer units forms an angle of 90° with $\vec{a}$, whereas the activity vector of any task that utilizes either memory or floating point units—such as $\vec{b_2}$—forms a smaller angle with $\vec{a}$.

The components of all activity vectors are positive or zero, so no two vectors can form an angle greater than 90°, since they all point into the same orthant. Therefore, we choose the task whose activity vector $\vec{b_j}$ with $j \in \{1,..,c\}$ forms the biggest angle with $\vec{a}$:

$$j = \arg\max_{i=1}^{c} \angle(\vec{a}, \vec{b_i}) \tag{4.1}$$

With $\angle(\vec{x}, \vec{y}) = \arccos \frac{\vec{x} \cdot \vec{y}}{|\vec{x}||\vec{y}|}$ we transform this to:

$$j = \arg\max_{i=1}^{c} \arccos \frac{\vec{a} \cdot \vec{b_i}}{|\vec{a}||\vec{b_i}|} \tag{4.2}$$

Since for $0° \leq \alpha \leq 90°$, $\arccos\ \alpha$ is strictly falling, this is equivalent to:

$$j = \arg\min_{i=1}^{c} \frac{\vec{a} \cdot \vec{b_i}}{|\vec{a}||\vec{b_i}|} \tag{4.3}$$

$|\vec{a}|$ is a constant, so we can omit it from the equation:

$$j = \arg\min_{i=1}^{c} \frac{\vec{a} \cdot \vec{b_i}}{|\vec{b_i}|} \tag{4.4}$$

We base our scheduling policy on a modified version of this formula. Instead of using the Euclidean length $||\vec{b_i}|| = \sqrt{\sum_{k=1}^{n} b_{i,k}^2}$, where $b_{i,k}$ is the k-th component of vector $\vec{b_i}$, we use the Manhattan length $||\vec{b_i}||_{\text{manh}} = \sum_{k=1}^{n} |b_{i,k}|$. Since all components of our vectors are positive, we can omit using absolute values. Hence, we choose the task with activity vector $\vec{b_j}$ to be scheduled next, if:

$$j = \arg\min_{i=1}^{c} \frac{\vec{a} \cdot \vec{b_i}}{\sum_{k=1}^{n} b_{i,k}} \tag{4.5}$$

Both, equations 4.4 and 4.5 lead to choosing a task that uses, as far as possible, different units than its predecessor, which is accomplished by the scalar product in the numerator of the fraction. Using the Manhattan length of $\vec{b_i}$ in the denominator instead of the Euclidean length leads to preferring activity vectors having medium values in many components to vectors with high values in some components and low values in other components, since for the former ones, the Manhattan length is, in particular, greater than the Euclidean length (see Figure 4.3). This is advantageous for our strategy, since scheduling several tasks causing medium utilization of the same units does typically not lead to hotspots, although the vectors of such tasks point in similar directions. In addition, determining the Manhattan length is computationally less expensive than determining the Euclidean length. (The latter involves calculating square roots.)

The policy of simple runqueue sorting uses a uniform view on activity vectors: As mentioned, calculating the angle between two vectors does not distinguish between individual vector components, but considers all components equally. Simple runqueue sorting avoids scheduling two tasks in a row that have high values for the same vector components, no matter what particular units these resources stand for. This has the advantage that the scheduling policy requires no knowledge about the topology and the actual temperature distribution of the chip, but can also lead to suboptimal scheduling decisions, as we will detail in the next paragraph.

Enhanced runqueue sorting

We also propose an enhanced form of runqueue sorting that considers the actual temperatures of the chip units for making scheduling decisions. As described in Section 4.2, the temperatures of the individual chip units cannot be obtained directly from the hardware. Therefore, we use a thermal model of the processor for estimating the temperatures.

As mentioned above, simple sorting assumes that tasks are exhausting their timeslice or at least consuming a significantly long portion of it, so that upon releasing the CPU, the temperature distribution of the processor chip corresponds to the task's characteristics. If a task uses only a small fraction of its timeslice and then blocks

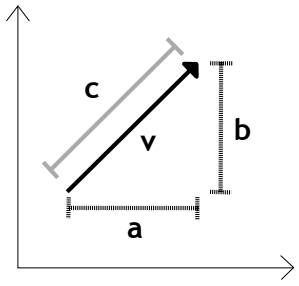


Figure 4.3: Manhattan length ($a + b$) and Euclidean length (c) of a two-dimensional vector $\vec{v}$

or voluntarily releases the CPU, the characteristics of this task are not crucial for the temperature distribution of the CPU, since during short periods of time, formerly cold units utilized by the task do not heat to the saturation point, and formerly hot units not utilized by the task do not have enough time to cool down.

Enhanced sorting can cope with this, since the thermal model considers the amount of energy each tasks dissipates in each unit, which is small if a task runs only for a short time. Thus, using a thermal model implicitly considers the actual period of time that a task spends on the CPU.

In addition, using a temperature model has the advantage of considering the physical properties of the chip, for example lateral heat spreading: Even a unit that the previously running task has not utilized can have an increased temperature, if it is located next to a unit that the task has utilized heavily. On the downside, using estimated temperatures for runqueue sorting is computationally more expensive than simple runqueue sorting, owing to the calculations required for the thermal model.

To consider temperatures on the chip, enhanced sorting needs to associate the individual vector components with chip units. Therefore, enhanced runqueue sorting uses a non-uniform view on activity vectors. In addition, a thermal model is always microarchitecture-specific. Although the thermal model need not be integrated into the scheduler, but can be implemented as a separate component that supplies the scheduler with the temperature values, which maintains the portability of the scheduling policy itself, the need for a thermal model increases the effort required for applying enhanced runqueue sorting to a new platform. In return, enhanced runqueue sorting offers the potential for more accurate scheduling decisions.

Thermal model For our purposes, a rather simple thermal model is sufficient. As mentioned before, our approach of temperature-aware scheduling is best-effort. Thus, wrongly estimated temperatures only lead to performance degradation, but not to hardware damage: If our scheduling policy takes a “wrong” decision caused by an inaccu-

racy of the temperature model, which would lead to a hotspot, hardware mechanisms engage throttling, so the only consequence is reduced performance.

In addition, our proposed scheduling policy does not rely on absolute accuracy, but works by comparing temperatures. Therefore, a temperature model that delivers the right trends (whether a unit is currently rather hot or rather cold) is already beneficial.

Rather than on being exact to the point, the focus our model is on being computationally inexpensive in order to limit the overhead introduced by temperature estimation. In the following, we want to describe the thermal model we use briefly.

Similar to Skadron et al. [SSH+03], we use a *dynamic compact model*. Compact models seize the fact that the phenomena of thermal capacitance and thermal conductance can be seen analogously to the respective electric phenomena, electric capacitance and electric conductance. Hence, a compact model represents the microprocessor and its heat sink by a network of thermal capacitors and resistors. For the reasons stated above, we choose a simpler model than Skadron et al. and model only the different units of the CPU plus the heat sink, omitting interface materials and heat spreader. Our model delivers an estimated temperature for each chip unit and is comparable to Temptor [HKK06], another adoption of Skadron's methodology to runtime temperature estimation. Our model is simpler than Temptor and thus less exact, but requires fewer calculations.

We derive the chip units of the processor from the floorplan and model each one as a thermal capacitor whose capacitance depends on the volume of the corresponding block on the chip, which we calculate by multiplying the area the block occupies on the floorplan with the chip height. In addition, we model the heat sink as one huge thermal capacitor. Each chip block is linked to each neighboring block via a thermal resistor with conductance proportional to the length of the border the blocks have in common. Also, each block is linked to the heat sink via a resistor whose conductance is proportional to the surface area of the block. The heat sink is linked to the surrounding air (which corresponds to ground in an electrical circuit) via a thermal resistor. Figure 4.4 shows a basic example of our thermal model for a hypothetical chip with four units.

While the processor is powered on, the chip units dissipate heat. As a consequence, the energy used to power the chip gets partially stored in the units (thermal capacitors), resulting in an increased temperature of the units. The temperature gradient between the chip units and the heat sink leads to a conduction of energy from the units to the heat sink via the thermal resistors. Analogously, the heat sink stores a part of this energy, resulting in increased temperature of the heat sink, which in turn causes an energy flow from the heat sink to to the surrounding air. In addition, there is also a lateral energy flow within the chip, meaning that energy is conducted from hotter areas to cooler ones.

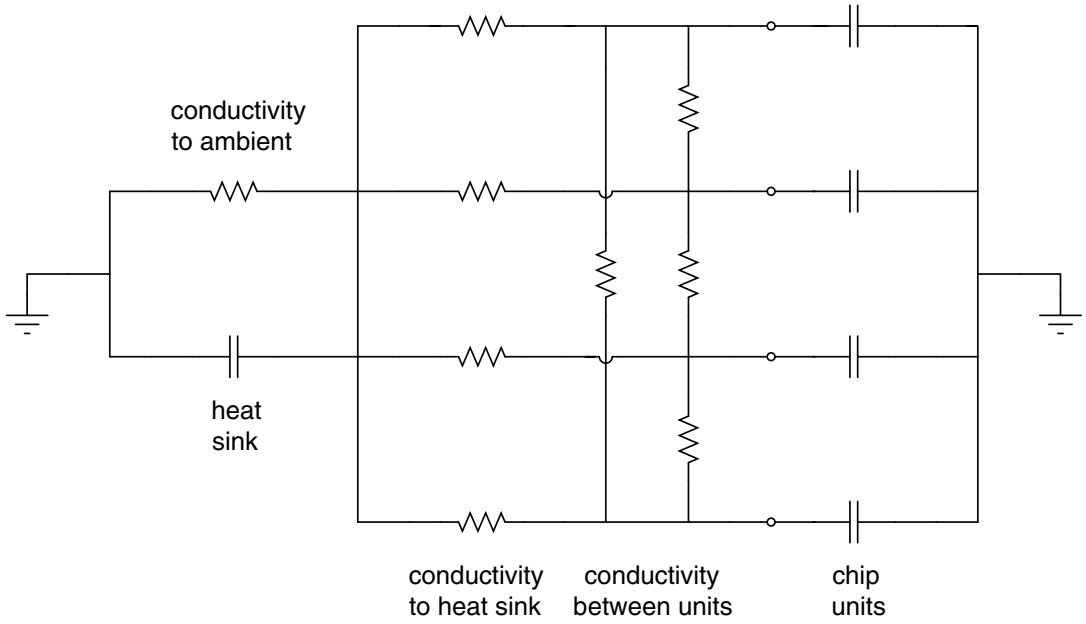


Figure 4.4: Thermal model of a simple chip with four units

While Skadron et al. uses computationally expensive numerical methods to solve the differential equations describing the network of thermal capacitors and resistors, we use simple discrete calculations.

We estimate the amount of energy that is dissipated in a chip block by using activity information. We follow the approach of Isci and Martonosi [IM06] and approximate the power dissipation of each block by a fixed part P_{base} that is independent of activity and models leakage, and a dynamic part that is proportional to unit utilization and models the power consumed by switching activity. We calculate the dynamic part by multiplying the maximal dynamic power P_{dyn} of the block with the utilization u (which ranges between 0 for no utilization and 1 for full utilization). For simplicity, we neglect the fact that leakage, and thus the units' power dissipation, is dependent on temperature. In summary, the amount of energy ΔE_{diss} dissipated during the time Δt becomes:

$$\Delta E_{diss} = \left(P_{base} + P_{dyn} \cdot u\right) \Delta t \tag{4.6}$$

The amount of energy ΔE_{cond} conducted between two components (two chip blocks, a chip block and the heat sink, or the heat sink and ambient air) during Δt depends on the conductance c of the thermal resistor connecting the components and on the temperatures ϑ_a and ϑ_b of the components.

$$\Delta E_{cond} = c\,(\vartheta_b - \vartheta_a)\,\Delta t \tag{4.7}$$

At each timer interrupt, we update the amount of energy stored in each of the thermal capacitors.

- We add the amount of energy that was dissipated in each chip block since the last timer interrupt to the block's capacitor according to Equation 4.6.
- We subtract the amount of energy that was conducted from the block to the heat sink from the block's capacitor and add it to the heat sink's capacitor. The amount of energy is determined according to Equation 4.7.
- For each two blocks adjacent to each other, we subtract an amount of energy corresponding to lateral heat transfer from the hotter block's capacitor and add it to the cooler block's capacitor, also according to Equation 4.7.
- We subtract from the heat sink's capacitor the amount of energy that was conducted to the ambient air since the last timer interrupt, again according to Equation 4.7.

After updating the energy stored in each capacitor, we calculate the new temperatures of the components, which are proportional to the respective amounts of energy stored in the components and the components' heat capacities.

Scheduling policy The basic idea of enhanced runqueue sorting is to choose a task to be scheduled next that utilizes chip units having a low temperature and does not utilize units having a high temperature. For deciding whether a unit's temperature is currently high or low, we compare the current temperature of the unit to the average temperature of the unit observed over the past timeslices. For this purpose, we calculate an exponential moving average over the temperature of each unit (as delivered by our thermal model).

Let $\vec{m}$ be a vector containing the average temperatures of all units and $\vec{t}$ a vector containing the current temperatures of the units. The difference $\vec{t}-\vec{m}$ is a vector with positive components for units that are currently hot and negative components for units that are currently cold.

Let again $\vec{b_i}$ with $i \in \{1,..,c\}$ be the activity vectors of the tasks we consider for scheduling for the next timeslice. When choose the task with activity vector $\vec{b_j}$ if:

$$j = \arg\min_{i=1}^{c}(\vec{t}-\vec{m}) \cdot \vec{b_i} \tag{4.8}$$

If a unit is currently hotter than its average temperature, scheduling a task that uses this unit is discouraged: The corresponding component of the temperature difference $(\vec{t}-\vec{m})$ is positive; multiplication with a nonzero utilization from vector $\vec{b_i}$ yields a positive contribution to the scalar product. On the other hand, if a unit is currently

colder than its average temperature, scheduling a task that uses this unit is encouraged: The corresponding component of the temperature difference is negative; multiplication with a nonzero utilization from vector $\vec{b_i}$ yields a negative contribution to the scalar product.

4.4.2 Activity balancing

Runqueue sorting is only effective if a runqueue contains tasks with different characteristics. In a multiprocessor system, the scheduler can influence the contents of the runqueues by migrating tasks between CPUs. In today's operating systems, this is done for the purpose of load balancing, i.e., equalizing runqueue lengths by moving tasks from long runqueues to shorter ones.

We use task migration to create the prerequisites for runqueue sorting. For example, if one runqueue consists solely of integer tasks and another runqueue consists solely of floating point tasks, we migrate some integer tasks to the second queue and some floating point tasks to the first queue.

Activity balancing is beneficial if done between processors that are physically different chips, as well as between different cores of a multicore chip, since it makes sure that for each location (chip or core), tasks having different characteristics are available. For logical processors of a simultaneously multithreaded processor, however, a different strategy makes sense, since several logical processors dissipate heat at the same physical location. We will describe this strategy in Section 4.4.3.

In previous work, we have proposed *energy balancing* for equalizing the power consumptions of CPUs by migrating tasks depending on the energy they consume during a timeslice [Mer05, MB06]. We propose a similar algorithm, *activity balancing*, to equalize the utilization of chip units between the CPUs. In contrast to energy balancing, where decisions upon task migrations are steered by just one parameter (power consumption), activity balancing considers multiple parameters, namely the components of the tasks' activity vectors.

To solve this multi-dimensional optimization problem heuristically, we construct a scalar measure as the basis for deciding whether the migration of a task is beneficial or not. This measure, which we call *thermal stress*, describes whether the tasks contained in a runqueue use mainly the same units or not. The goal of activity balancing is to keep the thermal stress of all runqueues as low as possible by migrating tasks between queues.

We define the thermal stress of a CPU as the sum of the average utilization of all units whose utilization is greater than a constant l. If the average utilization of a unit is greater than l, this means that too many tasks are utilizing the unit too heavily, and runqueue sorting cannot always find a task that does not utilize the unit.

Using the average utilization as a metric assumes that the tasks in a runqueue have equal timeslice lengths. If different tasks have timeslices of different lengths, using a

weighted averaging function that gives higher weights to tasks with longer timeslices would be appropriate.

Since for the calculation of thermal stress, no special meaning is associated with the individual vector components, activity balancing has a uniform view on activity vectors.

Definition. *Let n be the dimension of the activity vectors, p the number of tasks in a runqueue, and $\vec{b}_i, i \in \{1,\ldots,p\}$ the activity vectors of the tasks in the queue. Let $\vec{m}$ be the average of the activity vectors $\vec{b}_i$:*

$$\vec{m} = \frac{1}{p}\sum_{i=1}^{p}\vec{b}_i \tag{4.9}$$

We define the thermal stress *s of the queue formally as:*

$$s = \sum_{j\in\{1,\ldots,n\}:\ m_j > l} m_j \tag{4.10}$$

This way, units that are used only by few tasks, or used by many tasks, but only to a low degree, do not contribute to thermal stress. Hence, migration of a task that utilizes a certain unit heavily to a runqueue whose tasks do not utilize the unit has the potential of reducing the thermal stress of the source queue while not increasing the target runqueue's thermal stress.

Activity balancing works by migrating a task from one runqueue to another if this either decreases the thermal stress of both queues, or decreases the stress of one queue, while the stress of the other queue remains constant. At the same time, we ensure that activity balancing does not cause load imbalances by migrating a task back for compensation if necessary. In this case, we select a task whose migration does not increase the thermal stress of any of the two queues. Vice versa, we also consider thermal stress when doing load balancing. If the need for a task migration arises because of a load imbalance, we select a task whose migration does not increase the thermal stress of the queues the task is migrated between for resolving the imbalance.

The parameter l determines the average level of utilization that is acceptable for a chip unit. The lower the value of l, the more difficult it is for activity balancing to find a target runqueue for migrations: If a task x utilizes a certain unit to a high degree, and we want to migrate task x to another runqueue without increasing the target queue's thermal stress, the target queue needs to contain enough tasks that utilize the unit concerned to a low enough degree to offset the contribution of task x in order to keep average utilization below l.

On the other hand, the higher the value of l, the more difficult it is for runqueue sorting to avoid hotspots: For each unit that is utilized heavily by a task, runqueue sorting requires the availability of at least one other task that does not utilize the unit, since this permits to schedule tasks using and not using the units alternatingly. From

this viewpoint, a value as low as $l = \frac{1}{2}$ would be optimal, since if the average utilization of a unit is $\frac{1}{2}$, this means that there are as many tasks that use the unit as there are tasks that do not.

For our experiments described in Section 4.6, we use a value of $l = \frac{2}{3}$, which is a compromise that enables activity balancing both to find target runqueues for migration, and to create runqueues that are reasonably suitable for runqueue sorting. A value of $l = \frac{1}{2}$ turned out to be too restrictive and to prevent migrations in many situations, when no configuration could be found that keeps unit utilization below $\frac{1}{2}$.

4.4.3 Activity unbalancing

Multithreaded CPUs offer multiple logical processors that make use of the hardware units of the same physical chip. Typically, multithreaded processors show even higher power densities and temperatures than single-threaded processors, since feeding the execution engines from several instruction streams leads to higher utilization of the chip units [YSBZ05].

Gomaa et al. [GPV04] has shown that regarding thermal constraints, multithreaded chips can be used most efficiently if siblings execute tasks that use different units (and hence possess different activity vectors). With our approach, we want to verify that the principle of running tasks with different characteristics together can be achieved in a real operating system by using automatic task characterization (activity vectors) and a suitable scheduling policy (activity unbalancing).

From the operating system's point of view, the logical processors of a chip behave like individual processors, and most operating systems treat them this way in many respects. In particular, scheduling decisions happen independently for each logical processor, and operating systems that provide CPU-local runqueues typically provide a separate runqueue for each logical processor.

Scheduling decisions for different logical processors do not necessarily happen at the same time, since tasks on different logical processors can block or release the CPU at different points in time. Synchronizing scheduling decisions to enforce that siblings do not execute tasks with similar activity vectors simultaneously would require coordination between the logical processors. While this is possible and leads to a form of gang scheduling (see Chapter 5), we choose a simpler solution in this place. As we will discuss in Section 5.8.7, a solution using coordination of scheduling decisions is also viable, but is more complex and does not have any additional benefits in the context of avoiding hotspots.

We solve the problem of running tasks with different characteristics simultaneously without using synchronization by using task migrations to arrange the runqueues of siblings in a way that the tasks in a runqueue all use different units than the tasks in the other siblings' runqueues.This solution is viable if the number of siblings per physical processor is small in comparison to the number of chip units considered, as is the

case, for example, with Intel's Hyper-Threading [MBH+02] employed on NetBurst, Core i7, and Atom processors.

As an example, on a two-way multithreaded processor, after arranging the runqueues, one queue might contain only cache intensive tasks and memory intensive tasks, and the other queue only floating point tasks and integer tasks. This way, it is unimportant whether scheduling decisions happen independently for the siblings, since no matter which task is chosen, it is guaranteed to use different units than the task running on the sibling.

For sibling processors, our goal is the opposite of what we want to achieve for physically different processors with activity balancing. For runqueues belonging to physically different processors, we want to spread tasks with similar characteristics over several runqueues in order to avoid having too many similar tasks on one physical processor. For runqueues belonging to siblings on the same physical processor, however, it is desirable *not* to spread tasks with similar characteristics over the runqueues, but to collect them in one runqueue in order to avoid the possibility that similar tasks are scheduled on different siblings at the same time.

Therefore, we must apply a policy different from activity balancing between siblings. Our goal is to distribute all available tasks to runqueues in a way that the tasks in one runqueue use different units than the tasks in all other runqueues. Thus, we propose a strategy for *activity unbalancing*.

A naïve idea for activity unbalancing would be to do the exact opposite of activity balancing, i.e., to use migrations that increase thermal stress. But increasing thermal stress is not necessarily our goal; we do not necessarily want to have tasks that all use the same units in a logical processor's queue, merely tasks that use units different from those used by tasks in the siblings' queues. This is a weaker requirement, and thus allows a policy that is more likely to find a suitable assignment of tasks to processors than a policy that maximizes thermal stress.

The policy we propose is based on a measure we call *diversity*. The diversity of two runqueues q_1 and q_2 denotes whether the tasks in q_1 mainly use the same units as the tasks in q_2 or not.

Definition. *Let n be the dimension of the activity vectors and $\vec{m}$ the average of the activity vectors of the tasks from q_1, as defined by Equation 4.9, and $\vec{o}$ the average of the activity vectors of the tasks from q_2, respectively. The* diversity $d(q_1,q_2)$ *is then defined by the following equation:*

$$d(q_1,q_2) = \sum_{j=1}^{n} |m_j - o_j| \tag{4.11}$$

Hence, if two siblings utilize a chip unit to a different degree, i.e., most tasks on one sibling utilize the unit heavily, while most tasks on the other hardly utilize it, this causes a high contribution to the diversity of the siblings. On the other hand, siblings

with runqueues that contain tasks that, on average, utilize the same units, have a low diversity.

Analogously to activity balancing, activity unbalancing works by migrating a task from one sibling to another, if this increases the diversity of the siblings. In systems consisting of multiple multithreaded chips, we perform activity unbalancing between runqueues belonging to siblings, but activity balancing between runqueues belonging to different physical processors, since this ensures that there are tasks with different characteristics available on each physical processor to distribute between the logical processors.

Besides running tasks with different characteristics simultaneously on siblings, we additionally perform runqueue sorting on each sibling. This is beneficial, since typically, the number of units on a processor is greater than the number of logical processors. Hence, even if we distribute the tasks to the runqueues in a way that the logical processors always use mutually different units, the units that the tasks in a particular runqueue are using may still differ from task to task, so that runqueue sorting is still beneficial.

4.5 Implementation

4.5.1 Activity vectors

As mentioned at the beginning of this chapter, we chose the Intel NetBurst architecture as our evaluation platform and thus implemented activity vectors for this platform. Our implementation is based on a Linux 2.6.16 kernel

As described in Chapter 3, we enhanced the `task_struct` data structure that Linux uses to maintain task state by fields describing the task's activity vector. For determining activity vectors, we implemented the mechanism described by Isci and Martonosi [IM03] to obtain unit utilization by evaluating performance monitoring counters of the Pentium 4. The mechanism delivers utilization for 22 chip units. We enhanced the mechanism for supporting multithreaded processors, which involves additional multiplexing of counters, since Hyper-Threading for the NetBurst architecture features one set of performance monitoring counters per physical processor, shared by two logical processors.

Owing to the limited number and configuration options for the performance monitoring counters present in the Pentium 4, Isci and Martonosi proposes a four-way rotation of counters for a single-threaded processor. To support multithreading, we extend the four-way rotation to an eight-way rotation, which in addition to switching between the four configurations proposed by Isci and Martonosi, switches between counting events for the first or the second sibling.

To capture task characteristics accurately, multiplexing has to happen at a higher frequency than the frequency of scheduling. The standard timeslice length in Linux

is 100ms. We chose to multiplex at 1ms intervals, especially since we also do an evaluation of shorter timeslice lengths (Section 4.6.6).

The counters for floating point, trace cache, and front side bus events are not able to distinguish between logical processors, but count events regardless from which sibling they originate. For these events, we divide the event count and assign half the events to each sibling. This introduces error into the task characterization. Similarly, multiplexing introduces error, since we cannot observe all events all of the time (cf. Chapter 3). As our evaluation shows, the characterization obtained this way is still accurate enough for the scheduling policies to yield significant benefits. In addition, the inaccuracy is no inherent problem; the accuracy of activity vectors could be improved if the hardware provided more suitable monitoring facilities that report utilization caused by each logical processor without requiring multiplexing.

We also implemented energy estimation based on unit utilization as proposed by Isci and Martonosi and temperature estimation using a compact model as described in Section 4.4.1 to obtain temperatures for each chip unit, which is required for the extended version of runqueue sorting.

4.5.2 Runqueue sorting

We implement runqueue sorting by modifying the *O(1) Scheduler* used in Linux up to version 2.6.22. The O(1) Scheduler combines round-robin scheduling with priorities. The scheduler assigns more CPU time to tasks with higher priorities and schedules tasks with higher priorities before tasks with lower priorities. However, the scheduler avoids starvation of low-priority tasks by making sure that no high-priority task receives a new time quantum unless all lower-priority tasks have been scheduled in the time in between.

This is achieved by introducing two data structures for storing ready-to-run tasks, one containing tasks that have already exhausted their time quantum (the *expired array*) and one for tasks that have not yet exhausted their time quantum (the *active array*). Both arrays consist of linked lists of tasks, one list for each of the 140 priorities Linux supports (multilevel queueing; see Figure 4.5). When making a scheduling decision, the scheduler chooses the first task of the linked list with the highest priority from the active array. After executing, the task is enqueued in the linked list of the expired array that corresponds to its priority. When the list with the highest priority is empty, the scheduler resorts to the second highest priority, and so forth. When all lists of the active array are empty, the scheduler exchanges expired and active array.

To support runqueue sorting, we modify the policy for choosing the next task to schedule. Instead of choosing the first task from the queue with the highest priority in the active array, we look at the first c tasks in the queue with the highest priority in the active array. If the queue with the highest priority contains less than c tasks, we also consider tasks from the queue with the second highest priority and so forth. In our implementation, we chose $c = 4$. We choose the task that suits best according

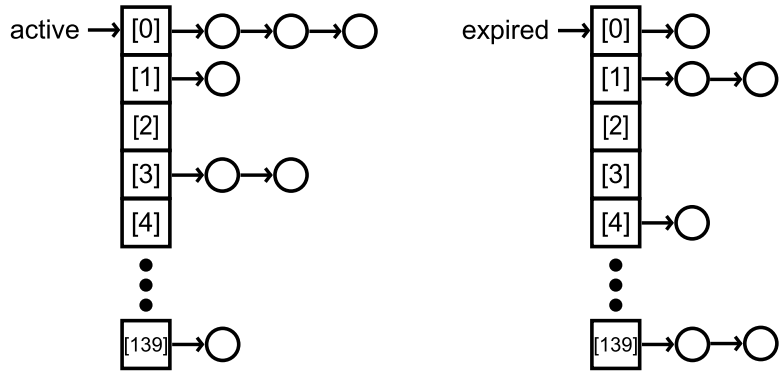


Figure 4.5: Active and expired array of the Linux O(1)-Scheduler

to our metric (Equation 4.5 or 4.8), execute it for one timeslice, and append it to the expired array. Like for the activity vectors, we use fixed-point arithmetic for all metrics derived from activity vectors.

Considering tasks from lower priority queues for selection despite higher priority tasks being present softens the priority scheme of Linux. However, it increases the chances that a suitable task is found if there are only few tasks of a given priority. In addition, priorities in Linux are already designed to be soft priorities only. Native Linux also schedules a lower priority task although a higher priority task is in the ready state, if the low priority task is it the active array and the high priority task is in the expired array.

4.5.3 Activity balancing/unbalancing

For implementing activity balancing between physical processors and activity unbalancing between siblings, we modified Linux's load balancer to consider the metrics described by Equations 4.10 and 4.11 in addition to load. Although Linux does assign different timeslice length to tasks of different priorities, for simplicity, we use an unweighted average to calculate thermal stress (cf. Section 4.4.2). This is justified, since our focus is on non-blocking workloads that typically run at the same priority level.

We take advantage of the *scheduler domain* hierarchy that Linux uses to represent the processor topology of the system. The hierarchy defines groups of CPUs at different levels, e.g. groups comprising all siblings of the same physical processor, groups comprising all cores on a die, groups comprising all processors residing on the same NUMA node, and so on. Within groups consisting of siblings, we perform activity unbalancing, and migrate tasks if this increases diversity, but between CPUs not be-

longing to the same group of siblings, we perform activity balancing, i.e., migrate tasks if this decreases thermal stress.

Linux's load balancer checks periodically for load imbalances between the runqueues. We supplement these periodic checks with additional checks for opportunities to reduce thermal stress between physically different processors by activity balancing, or increasing the diversity of sibling processors by activity unbalancing.

Upon detecting an imbalance, Linux invokes a function for migrating tasks in order to resolve the imbalance. This function searches for candidate tasks that are suitable for migration according to various criteria, for instance, that the task is not currently running, and is expected to have little data in the processor's cache. We extend this balancing function to also reduce thermal stress and increase diversity. For physically different processors, we add to the criteria for a task migration the requirement that thermal stress of either the source or the target runqueue be reduced by the migration (activity balancing). For sibling processors, instead we add the requirement that the diversity of the runqueues be increased (activity unbalancing).

4.6 Evaluation

4.6.1 Setup and methodology

We evaluated our implementation with a set of workloads composed of programs taken from the SPEC CPU 2006 suite [Hen06], a benchmark suite consisting of several compute intensive integer and floating point benchmarks derived from real user applications. Appendix A gives an overview of the programs from the SPEC CPU suite.

As described in Section 4.2, the temperature sensors deployed on today's chips are not suitable for verifying the impact of our approach, namely the reduction of hotspots. Temperature measurement via external sensors such as laser thermometers is not applicable either, since it is not possible to operate the chip without a heat sink attached, and hence the chip is not accessible for temperature measurements.

For these reasons, we used a combination of real hardware and simulation. We let our modified Linux kernel run on real physical hardware. We used an IBM xSeries 445 eight-way multiprocessor system (eight Pentium 4 Xeon Gallatin processors with 2.2GHz each). The system consists of two NUMA nodes with four two-way multithreaded processors on each node.

For investigating temperature, we resort to simulation with the HotSpot [HSS+04] temperature simulator. During the test runs, we sampled the utilization of each chip unit using performance monitoring counters. From unit utilization, we estimated the power consumption of each unit. Afterwards, we used the power estimates as input to the HotSpot simulator, with which we performed offline temperature simulation to study the effects of our policies on temperature distribution.

Although we use performance monitoring counter readings both to guide our scheduling policies and to feed the temperature simulator used for evaluation, our methodology is still viable for evaluating the benefits of our policies. Our goal is not to prove that unit utilization and power consumption can be inferred from performance counters (this has already been done before [BWWK03, IM03]), but to show that if unit utilization is known, this knowledge can be used to influence temperature distribution.

Since we had no floorplan for the Pentium 4 Xeon Gallatin processor at our disposal, we performed temperature simulation based on the floorplan of a Pentium 4 Northwood processor, which has the same feature size and also the same amount of L1, L2, and trace cache, but lacks the L3 cache of the Gallatin processor. Since the power consumption of the L3 cache is moderate in relation to the area the cache occupies, the L3 cache is no hotspot and can therefore legitimately be neglected.

For evaluation, we sampled unit utilization at millisecond granularity. To limit the amount of data and to shorten the time required for the offline temperature simulations while at the same time capturing long-term behavior, we only consider every 100th second of the runtime for our evaluation. Thus, during a test run, the programs ran 99 seconds unobserved, then, during the following second, we took 1000 samples, let the program run for 99 seconds again, and so forth.

4.6.2 Overhead of activity vectors

We evaluated the overhead of our implementation of activity vectors. As already mentioned, the method of Isci and Martonosi for determining unit utilization requires multiplexing counters, i.e., saving and re-loading counter values periodically. Since for SMT chips, in addition, all counters have to be multiplexed between the two siblings, we expect a non-negligible overhead: While we measured an overhead of 140 cycles for reading a performance monitoring counter via the optimized `rdpmc` instruction, re-writing a configuration register via the `wrmsr` instruction, which is necessary for multiplexing, amounts to 950 cycles on our test system.

We measured the total number of cycles required for determining unit utilization (reading performance counters, calculating utilization, and re-configuring the multiplexed counters) by reading the time stamp counter at the beginning and at the end of our activity monitor routine. The overhead amounts to 33,000 cycles, or 1.5% of the cycles of a timer tick.

We also measured the resulting overhead for applications by executing the SPEC CPU benchmarks with activity vectors disabled and again with activity vectors enabled. Averaged over all benchmarks, we measured an increase in runtime of 1.4% when we enabled activity vectors. When we disabled multiplexing and just read the performance monitoring counters every millisecond, this increase was nearly halved to 0.8%. This strongly suggests that processors featuring a more suitable set of performance monitoring counters would reduce the overhead for characterizing tasks.

Furthermore, without multiplexing, there would be no need to sample counters every millisecond, like we do in our current implementation; sampling at every task switch would be sufficient. We measured a modified implementation that only samples counters at the frequency of task switches; here we observed no measurable increase in application runtime introduced by activity vectors. This indicates that with adequate hardware support, task characterization is possible "for free".

4.6.3 Runqueue sorting

To demonstrate the benefits of runqueue sorting, we selected two SPEC benchmarks that use complementary resources. `hmmer` uses mainly the integer units (integer ALUs and integer register file) as well as the L1 cache. When `hmmer` is running alone, the hottest unit on the chip is the integer register file. `namd` uses mainly the floating point units; when `namd` is running, the hottest unit is the floating point register file.

We started three instances of `hmmer` and three instances of `namd` simultaneously. Since we wanted the tasks to run on one CPU together, we disabled all CPUs of the system but one and also disabled simultaneous multithreading.

Figure 4.6 shows the effect of runqueue sorting. The figure depicts the course of temperature over time for the floating point register file, which reached the highest temperature of all units during the test.

Without runqueue sorting, the order in which the tasks are scheduled is arbitrary. In the most unfavorable situation, which is shown it the top half of the figure, three timeslices in a row are assigned to the three instances of `namd`, followed by three timeslices assigned to `hmmer`. Whenever an instance of `namd` is running, the temperature of the floating point registers increases, whereas whenever `hmmer` is running, the temperature of the floating point registers decreases. Since a timeslice is 100ms long, temperature rises/decreases for 300ms, respectively.

With runqueue sorting enabled, the scheduler arranges the tasks in a way that whenever an instance of `namd` has been executed for one timeslice, an instance of `hmmer` gets scheduled during the next timeslice. The effect of this can be seen in the bottom half of the figure: The time during which the floating point register file's temperature increases is one timeslice at most, since after this timeslice, the scheduler selects a task that does not utilize the floating point units. Therefore, the temperature of the register file does not rise as high as it does without runqueue sorting. The effect on the other units' temperatures is similar.

The benefits of runqueue sorting become visible in a histogram that displays the frequency of occurrence for the temperature values observed during the test run. Figure 4.7 shows histograms of the temperature of the floating point registers.

Without sorting (top histogram in Figure 4.7), two major spikes appear in the histogram, one around 60°C, and one around 82°C. When unit utilization is constant for a longer period of time, as is the case when many tasks with similar characteristics are scheduled successively, the units reach a steady state temperature (also compare

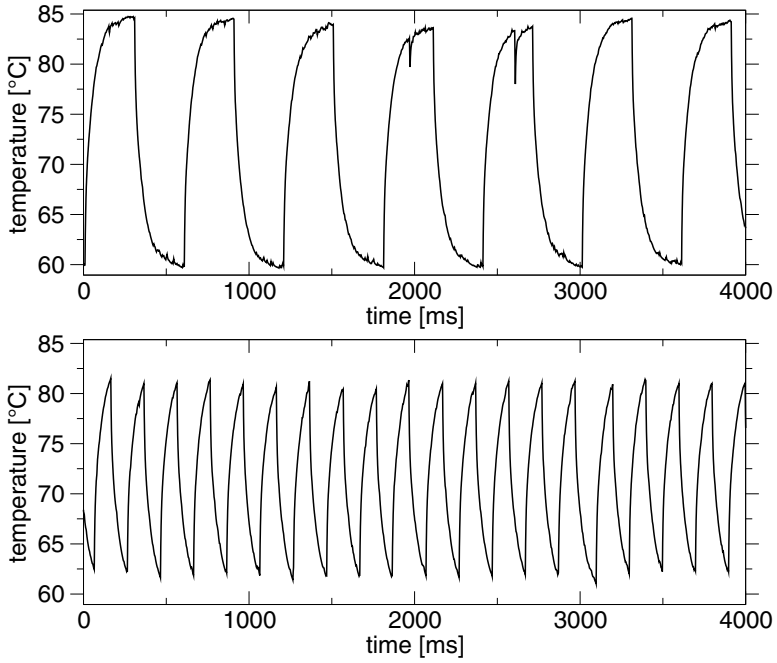


Figure 4.6: Temperature of the floating point registers with (bottom) and without (top) runqueue sorting

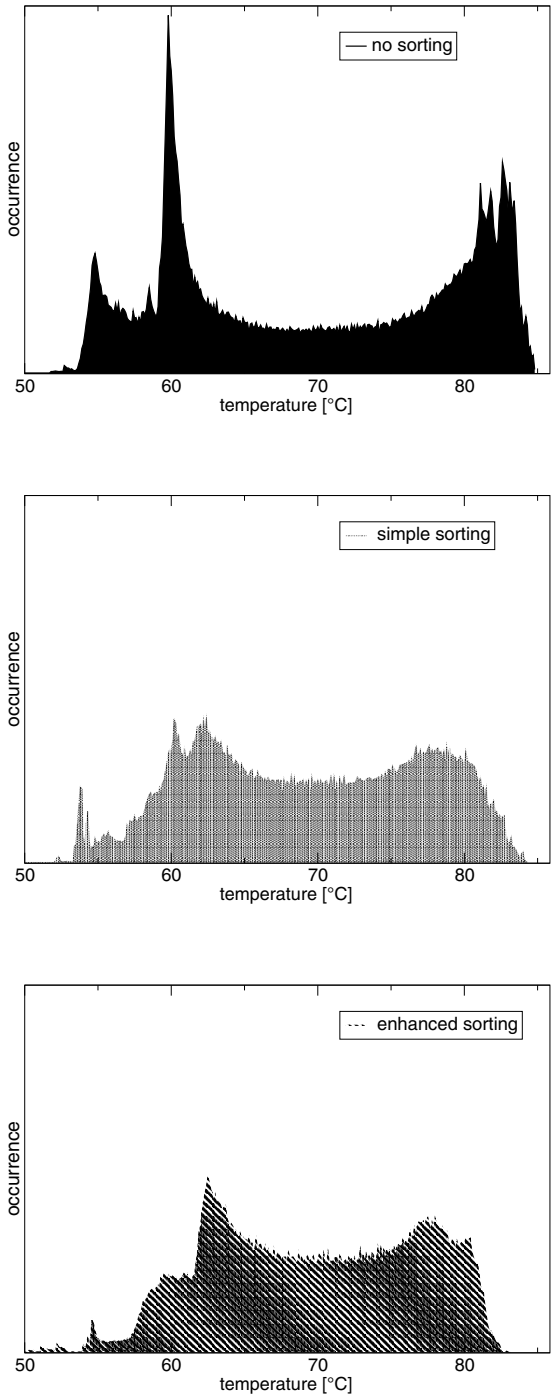


Figure 4.7: Temperature of floating point registers (hmmer + namd)

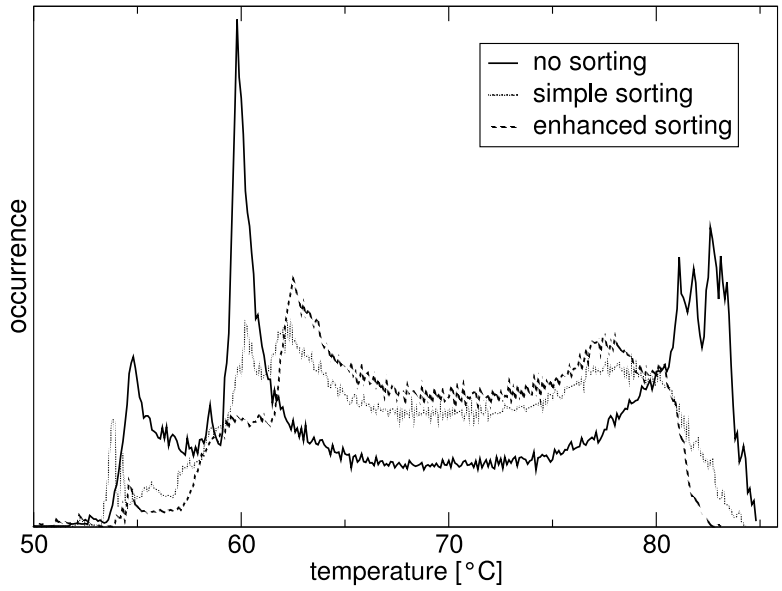


Figure 4.8: Temperature of floating point registers (hmmer + namd, combination of the histograms from Figure 4.7)

Figure 2.2). The spike at 60°C results from the floating point registers being inactive during a longer period of time, whereas the spike at 82°C results from the registers being active for a longer period of time. When runqueue sorting is enabled (bottom histograms in Figure 4.7), the spikes are diminished, and temperature is biased towards medium temperatures.

The effects of runqueue sorting on temperature become most apparent when overlaying the three histograms, as can be seen in Figure 4.8. To improve readability, the figure lacks the bars of the histograms and displays only the tops. As can be seen, runqueue sorting biases temperature towards the middle of the range, which in particular leads to high temperatures occurring less frequently, and also reduces the observed maximum temperature. The figure also shows that enhanced sorting guided by a thermal model is superior to simple sorting guided only by unit utilization.

In our experiment, without runqueue sorting, the temperature of the floating point register file was greater than 80°C for 25% of the time. With simple sorting, this percentage dropped to 9% and with enhanced sorting further to 6%.

For other combinations of tasks, runqueue sorting yields similar effects: If the tasks possess different characteristics regarding the utilization of certain chip units, the temperatures of these units get biased towards medium temperature ranges by runqueue sorting. Figure 4.9 shows histograms of the temperature for the data translation lookaside buffer (DTLB), which is the hottest unit when `gobmk`, a memory intensive bench-

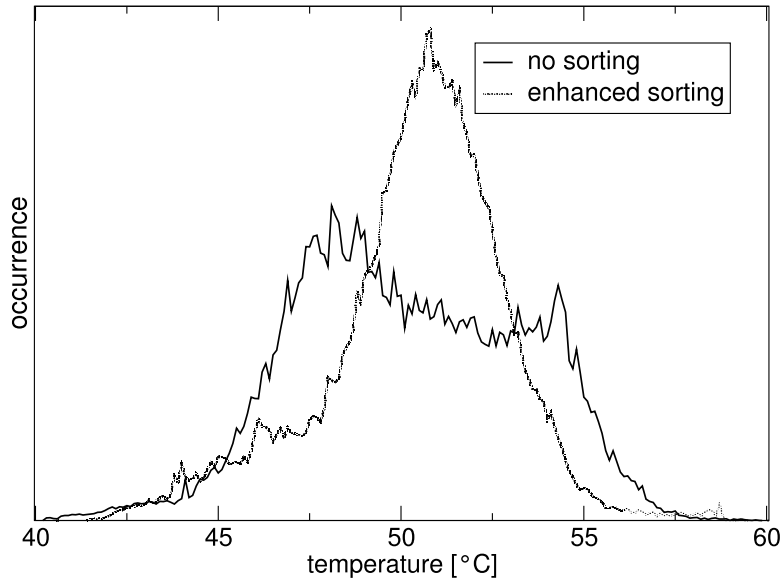


Figure 4.9: Temperature of DTLB (gobmk + leslie3d)

mark, is running in combination with `leslie3d`, which is also memory intensive, but does not stress the DTLB as much as `gobmk` does. Although the overall temperature of the DTLB is lower than that of the floating point registers in the previous test, a bias towards the medium range can also be observed. Since `gobmk` and `leslie3d` both use the DTLB to some degree, the spikes in the histogram are not as prominent as in the previous test.

For combinations of tasks that have similar characteristics, runqueue sorting is not beneficial. Figure 4.10 shows histograms for the temperature of the floating point registers when running `calculix` in combination with `milc`. Both benchmarks show high utilization of the floating point registers, although not as high as with `namd`. Therefore, the histogram shows only one spike around 70°C. Since the tasks have similar characteristics, runqueue sorting cannot reduce the temperature of the floating point registers and the histogram looks similar with sorting activated.

We also measured the overhead introduced by sorting. We compared our implementation of runqueue sorting to Linux's original scheduling policy. The runtime of the benchmarks increased by 1.3% with simple sorting and by 1.5% with enhanced sorting.

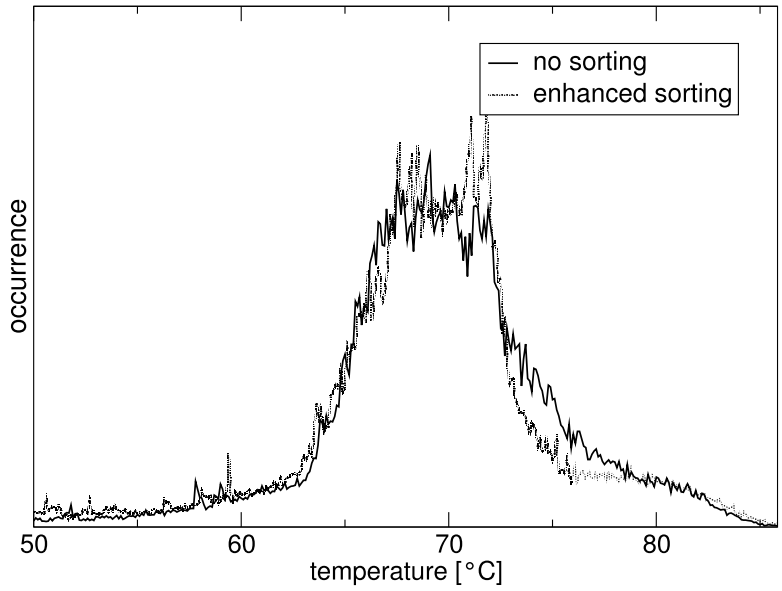


Figure 4.10: Temperature of floating point registers (calculix + milc)

4.6.4 Activity balancing

We tested activity balancing by running the same workload as described at the beginning of the preceding Section, but on all eight processors of the system (simultaneous multithreading disabled). Hence, we started 24 instances of each, `hmmer` and `namd`, which the load balancer distributed to the individual CPUs. To eliminate the possibility of buying the reduction of hotspots on one CPU with an aggravation on some other CPU, we observed the temperature of all eight CPUs.

We performed two runs, one with activity balancing disabled and one with activity balancing enabled. During both runs, we activated enhanced runqueue sorting. Since runqueue sorting depends on runqueues consisting of tasks with different characteristics, it is only beneficial if the load balancer distributes tasks to CPUs accordingly. The default Linux load balancer, however, is oblivious of the tasks' characteristics.

This becomes apparent in Figure 4.11, which displays histograms of the temperature values accumulated from all eight processors. Without activity balancing, the histogram looks similar to the one in the top of Figure 4.7, which resulted from a test run with runqueue sorting disabled. Runqueues containing too many instances of `namd` are responsible for this behavior. Only when runqueue sorting and activity balancing are combined, the desired effect eventuates.

Migrating a task to another processor introduces an overhead, since the task needs to warm up the cache of the new processor. However, activity balancing triggers task

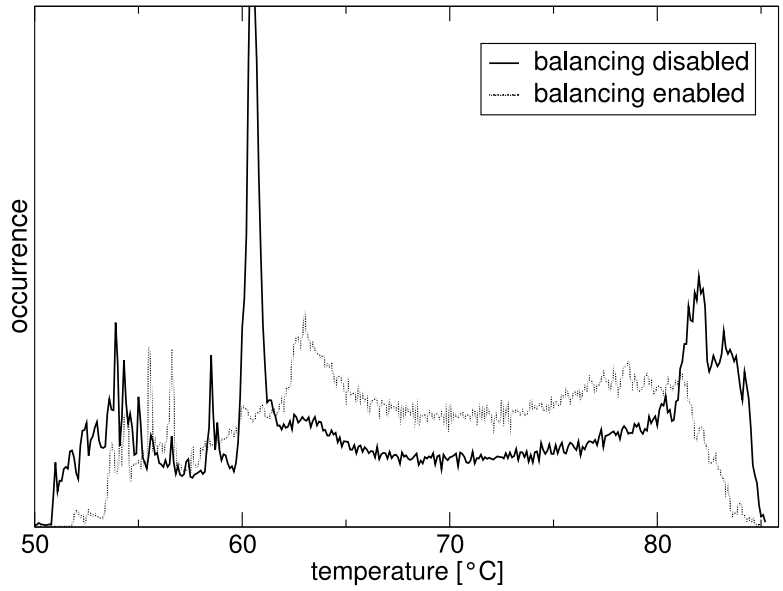


Figure 4.11: Temperature of floating point registers (8 processors)

migrations only when new tasks start or unblock, when existing tasks terminate or block, or when the characteristics of the running tasks change significantly. Otherwise, once the tasks are distributed to the CPUs according to their unit utilization, no further migrations are necessary. Thus, we measured only 0.3% overhead introduced by activity balancing.

4.6.5 Activity unbalancing

To test activity unbalancing, we enabled only one CPU in the system, but activated the processor's Hyper-Threading capability, which resulted in a system consisting of two logical CPUs. We ran one test with runqueue unbalancing enabled and one test with runqueue unbalancing disabled. Both times, we enabled runqueue sorting and ran four instances of `hmmer` and `namd`, respectively.

It is remarkable that with Hyper-Threading enabled, the histogram (Figure 4.12) looks completely different than with Hyper-Threading disabled. Instead of two spikes, one in the low and one in the high temperature ranges (compare Figure 4.8), with Hyper-Threading, the temperatures in the middle range dominate.

The reason for this becomes evident when looking at the course of temperature. Power consumption and temperature of the chip units are no longer determined by the order in which tasks are scheduled by one scheduler, but by the scheduling decisions of two independent schedulers. Figure 4.13 shows a typical section of the course

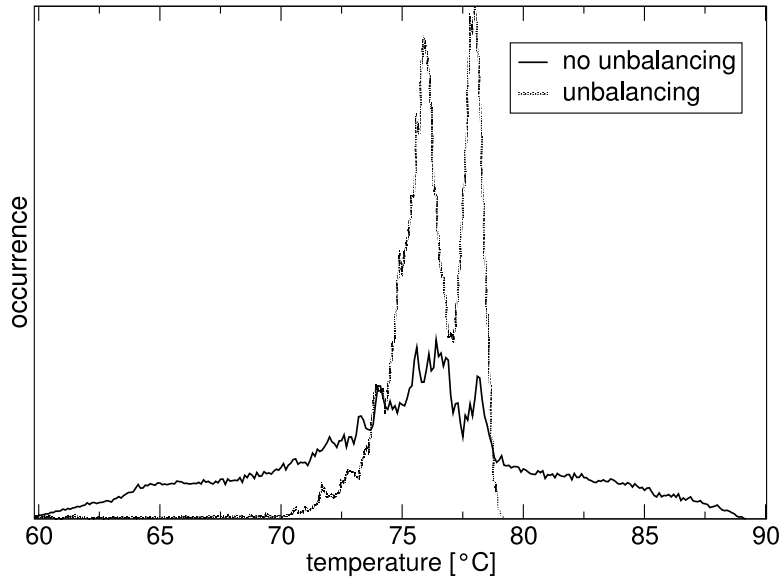


Figure 4.12: Temperature of floating point registers (Hyper-Threading)

of the floating point registers' temperature over time. Since task switches happen independently for both logical processors, this leads to situation in which none of the processors uses the unit as shown at point (1) in the figure, both processors use the unit (2), or one processor uses the unit and the other does not (3) and (4).

When one logical processor uses a unit and the other does not, the unit's temperature is in the middle range. Combined with the fact that it takes some time for temperature to increase or decrease after both processors start using or not using a unit at the same time, this yields the bias towards medium-range temperatures that can be observed in the histogram (Figure 4.12). Yet, temperatures in the lower and the higher ranges still occur.

Provided that the set of runnable task allows it, activity unbalancing ensures that tasks using complementary resources are always scheduled together on sibling processors. This avoids temperature spikes to the upper and lower ranges (like points (1) and (2) in Figure 4.13) and ensures that a situation as shown at points (3) and (4) dominates, where temperature is in the middle range.

The histogram in Figure 4.12 reflects this: With activity unbalancing enabled, temperature values are completely constrained to the middle range. This shows the benefits of activity unbalancing. Without unbalancing, the temperature of the floating point registers is higher than 80°C during 17.7% of the time. Activity unbalancing achieves that the temperature of the floating point register never exceeds 80°C in this scenario, so our policy succeeds in avoiding hotspots.

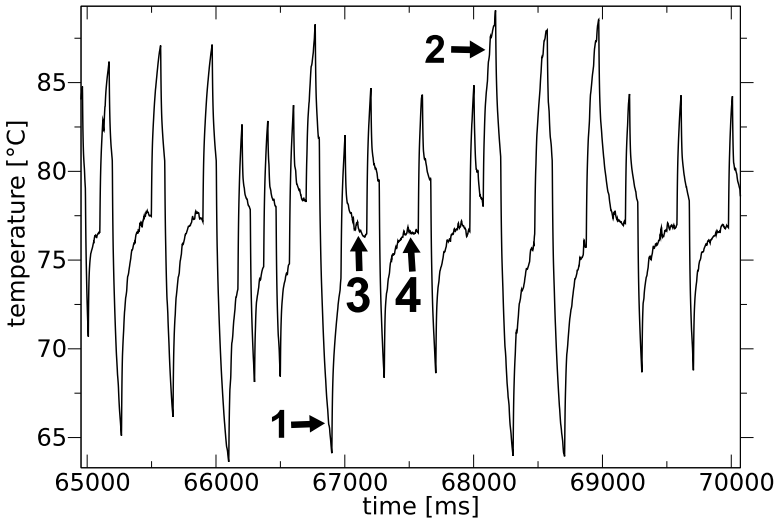


Figure 4.13: Course of temperature for floating point registers (Hyper-Threading)

Activity unbalancing is also advantageous in another respect: Running tasks that use complementary resources together also makes sure that the tasks running simultaneously on sibling processors obstruct each other less, since they do not compete as much for resources as would be the case when running tasks with similar characteristics together. This is the same principle symbiotic job scheduling [ST00] takes advantage of. Less resource contention means higher throughput; in our scenario, the runtime of the benchmarks decreased by 3.6% when activity unbalancing was enabled.

We verified that this is generally the case by test runs using other SPEC benchmarks: With one physical processor and Hyper-Threading activated, we ran six benchmarks simultaneously. The benchmarks were chosen at random out of the SPEC CPU2006 suite. Whenever one benchmark terminated, we started another benchmark, also selected at random, so there were always six benchmarks running simultaneously, but in arbitrary combinations.

We measured the runtimes of the benchmarks with activity unbalancing disabled and also recorded the order in which the benchmarks were started. After the first run, we enabled activity unbalancing and replayed the same sequence of benchmarks as before. A comparison of the runtimes showed an improvement (reduction of runtime) of 3.3%.

Improved performance by reduced resource contention comes as a byproduct of temperature-aware scheduling on multithreaded processors, since both goals, avoid-

setup	overhead [%]	maximum temperature fp regs [°C]
100ms	0.0	84.7
100ms, sorting	1.1	82.7
50ms	0.2	83.5
50ms, sorting	1.1	79.8
16ms	1.9	79.0
16ms, sorting	2.5	76.5

Table 4.1: Effects of different timeslice lengths in combination with runqueue sorting

ing hotspots and reducing contention for shared resources, are achieved by scheduling tasks using complementary resources together. We will discuss the avoidance of resource contention by vector-based scheduling in greater detail in Chapter 5.

4.6.6 Shortening timeslices

As mentioned in Section 4.3, Michaud and Sazeides [MS06] proposes using shorter timeslices to avoid thermal emergencies. The reason behind this approach is that with shorter timeslices, tasks utilizing different units on the chip are switched in quicker succession, so the units do not have enough time to heat up. On the other hand, shorter timeslices means higher task switching overhead caused by saving and restoring task context, switching the address space (which on some architectures means invalidating the TLB or virtually indexed caches), and cache misses caused by tasks evicting each other's data in the cache.

We argue that it is beneficial to shorten timeslices and at the same time apply temperature-aware scheduling. Even when timeslices are short, there is still the possibility that tasks with similar characteristics are scheduled successively, so scheduling can yield an improvement regardless of the timeslice length. Our measurements confirm that shorter timeslices and vector-based scheduling work best in combination.

Figure 4.14 depicts the temperature distribution for the scenario with three instances of `hmmer` and three instances of `namd` on one processor for different timeslice lengths and, optionally, with enhanced runqueue sorting. A comparison shows that 100ms timeslices in combination with runqueue sorting leads to lower maximum temperatures than 50ms timeslices without sorting. Also, 50ms and sorting is almost as good as 16ms without sorting. The most important observation, however, is that for any given timeslice length, runqueue sorting yields a considerable improvement.

Shortening timeslices comes with an overhead, since task switches introduce a penalty. The exact overhead introduced by shortening timeslices depends on the ar-

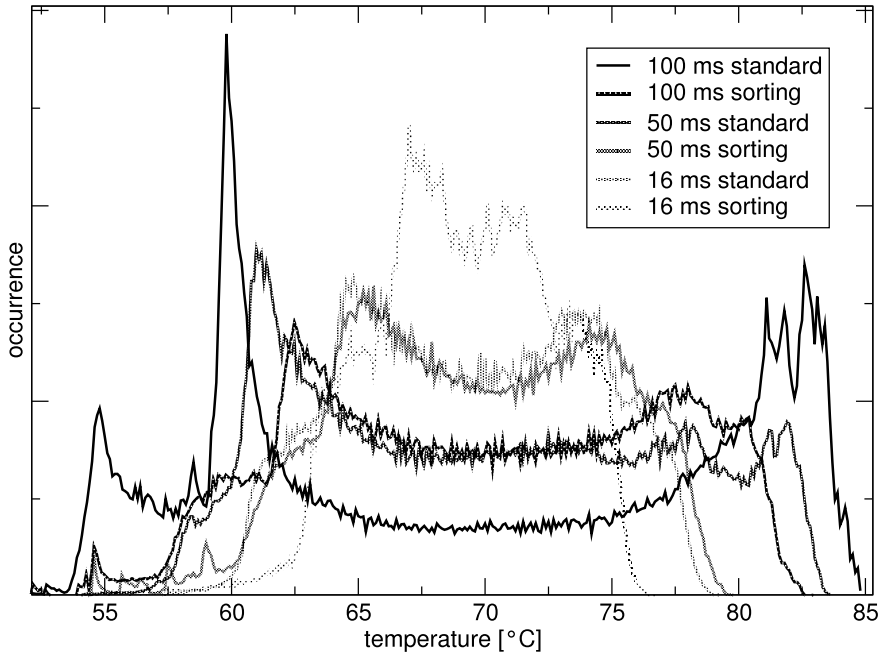


Figure 4.14: Temperature of the floating point registers with different timeslice lengths and scheduling policies

chitecture, which determines the cost of reloading task state, but also on the tasks themselves. For example, two tasks that both occupy large portions of the cache need to reload their working set into the cache on each task switch, whereas tasks occupying only a small portion of the cache are likely to retain their working set over the execution of the other task. The architecture determines the immediate cost for switching tasks (reloading task state), but also the size of the caches.

In our example scenario, the overhead of shortening timeslices is rather modest, since both `hmmer` and `namd` are compute-bound applications (cf. Chapter 5) with relatively small working sets. Table 4.1 shows the overhead and the benefits of different timeslice lengths and, optionally, runqueue sorting in comparison to standard Linux scheduling with 100ms timeslices. Two observations can be made: Firstly, reducing the timeslice length to 50ms causes only little overhead for this scenario, but has a noticeable effect on temperature. Secondly, further reducing the timeslice length to 16ms causes nearly twice the overhead of combining 50ms timeslices and runqueue sorting, while both setups lead to a comparable temperature distribution. Therefore, we suggest to use moderately short timeslices in combination with runqueue sorting.

Depending on the scenario, shortening timeslices can have a considerably bigger performance impact that in our example scenario. For demonstrating this, we constructed a worst-case microbenchmark that generates as much memory references as possible, and executed each SPEC benchmark together with the microbenchmark on one processor, using the standard Linux scheduler and timeslices of 100ms and then of 16ms. Linux schedules the tasks alternatingly for one timeslice each; thus, whenever the SPEC benchmark is scheduled for a timeslice, it needs to reload its working set into the cache, since the microbenchmark that has run during the previous timeslice has overwritten the cache.

In this setup, the runtime of `hmmer` increases by 3.3% when timeslices are shortened from 100ms to 16ms; for `namd` the increase is 2.4%. Other SPEC benchmarks, like the memory-bound `lbm` are affected much more; `lbm`'s runtime increases by 40% if timeslices are shortened from 100ms to 16ms. The reason for this tremendous increase is the cache architecture of our test system. Besides the L1, L2, and L3 cache of the Gallatin Processor, the IBM x440 also features a L4 system cache (XceL4 Server Accelerator Cache [HWW02]), which benefits applications like `lbm` that cause many L3 cache misses. Shorter timeslices lead to the caches being overwritten and the working sets of the tasks reloaded more often, which is especially costly for the L4 cache, owing to its large size of 64MiB.

We conducted the same test on a desktop system with a comparable Intel Northwood processor that just features L1 and L2 cache. Here, the `lbm` benchmark runs only 0.7% slower with 16ms timeslices. Since the desktop system lacks the deeper cache levels and the working set of `lbm` does not fit into the L1 and L2 cache, overwriting the caches more often has only little impact. The maximum slowdown we observed on the desktop machine for SPEC benchmarks was 3.5% for the `povray` benchmark.

This worst-case scenario shows that the cost of reduced timeslice lengths is task and architecture dependent, and a general comparison of the cost of vector-based, temperature aware scheduling and shortening timeslices cannot be made. The decision about the optimal timeslice length depends on the system concerned, the workload that is running, as well as on thermal and performance constraints. Determining the optimal timeslice length for a given setup is beyond the scope of this thesis.

However, runqueue sorting and reducing timeslice lengths are orthogonal mechanisms for balancing chip temperature. Since runqueue sorting causes only little overhead, it is beneficial to apply it regardless of the chosen timeslice length to reduce temperature further. In addition, short timeslices are only beneficial if tasks with different characteristics are assigned to a processor, which is accomplished by mechanisms like activity balancing.

4.6.7 Analysis

Our experiments show that information about the utilization of chip units, provided by activity vectors, can be used successfully for temperature-aware scheduling. Runqueue sorting improves the temperature distribution on the chip and reduces hotspots. Activity balancing creates the prerequisites for runqueue sorting in SMP systems. For multithreaded processors, activity unbalancing reduces hotspots and, in addition, increases performance.

The overhead introduced by vector-based scheduling has to be compared to the overhead that results when throttling has to be engaged to prevent overheating. The overhead introduced by throttling depends on the actual throttling mechanism, and on the trip temperature at which throttling starts. Both, mechanisms and trip temperatures, vary between different CPU types.

For instance, Intel's *thermal monitor 1* feature periodically disables the clock signal, effectively reducing the processor's frequency by 12.5% to 87.5%. The *thermal monitor 2* feature uses frequency scaling for reducing power and temperature; the penalty introduced by thermal monitor 2 depends on the frequency settings the hardware offers. In both cases, throttling results in a performance penalty that is considerably higher than the penalty introduced by our vector-based scheduling policies, which we measured to be 1.8% at most (1.5% introduced by runqueue sorting and an additional 0.3% introduced by activity balancing). This holds true even if, in addition, the cost of maintaining activity vectors by using performance monitoring counters not designed for this specific purpose is taken into account (1.4%, see Section 4.6.2).

As an example, we showed in Section 4.6.3 that for a scenario with tasks using complementary resources, runqueue sorting manages to reduce the percentage of time during which the hottest of the processor's units operates above 80°C from 25% to 6%. If the processor is not supposed to operate at more than 80°C, throttling has to be used to keep the temperature below 80°C, at the price of prolonged execution times. If we assume the CPU processes a task at half the original speed when throttled, without

runqueue sorting, 25% of the instructions have to be executed at half speed, which doubles the time required to process these instructions and increases total execution time by 25%. With runqueue sorting, the total execution time is only increased by 6% because of throttling, plus the additional 1.5% introduced by the overhead runqueue sorting causes. A program running 100s on an unthrottled processor thus runs 125s without runqueue sorting and 107.6s with runqueue sorting, which is a speedup of 14%. Even if a throttled processor is running at 75% maximum speed instead of 50%, the speedup is still 7%.

Depending on the processor, 80°C, which we have used as a reference temperature for illustrating the benefits of our approach, need not necessarily be a critical temperature. In the system we used, we were therefore not able to increase performance by decreasing temperature, since even without our improved temperature distribution, no throttling was engaged. Yet, with increasing power and integration densities, thermal problems can be expected to aggravate in the future. In addition, the lifetime and the reliability of a processor chip decrease with temperature. Increasing temperature by 10 to 15 degrees halves the lifetime of an electrical circuit [VWWL00, YC01]. Therefore, reducing hotspots is always beneficial.

Besides avoiding hotspots, our policies also have an impact on thermal cycling, i.e., the variation of chip temperature over time. As can be seen in Figure 4.6, runqueue sorting increases the frequency, but reduces the amplitude of the temperature swing.

Thermal cycling reduces chip lifetime by causing fatigue. The reason is that different materials of the die, the package, and the die interface show different degrees of thermal expansion. However, thermal cycling has only been investigated in the case of large-scale temperature cycles that result from powering the system on and off or from entering and exiting low-power states, and affect the entire package [JED08].

The effect of small-range, localized temperature cycles as those introduced by switching between tasks of different characteristics has not been well studied yet [SABR04], so it remains unclear whether increasing the frequency but reducing the amplitude of small thermal cycles aggravates or mitigates fatigue. Investigating the effects of thermal cycling is beyond the scope of this thesis, but, ultimately, should be considered when techniques like runqueue sorting are applied with the goal of increasing reliability.

4.7 Summary

The tasks running on a processor chip determine the chip's temperature distribution and the location of hotspots. This chapter has addressed the influence the operating system can take on the temperature distribution by scheduling tasks in a suitable manner. We have shown that activity vectors are a valuable input for a scheduler that aims at reducing hotspots. We have proposed three scheduling strategies that make use of activity vectors:

Runqueue sorting arranges the tasks in a processor's runqueue in a way that tasks using complementary resources are scheduled successively. This reduces thermal stress, since hot units can cool down when a task not using the units is scheduled. *Activity balancing* makes sure that runqueue sorting can be applied effectively in multiprocessor systems: By distributing tasks with similar characteristics over all processors of a system, we make sure that the composition of each runqueue is suitable for runqueue sorting. *Activity unbalancing*, in contrast, concentrates tasks with similar characteristics in the same runqueue. Activity unbalancing is beneficial if applied between SMT siblings, ensuring that tasks with different characteristics are running simultaneously, which reduces thermal stress and at the same time increases throughput.

Our experiments show that vector-based temperature-aware scheduling succeeds at reducing hotspots. Our scheduling policies considerably reduce the percentage of time during which the processor operates in high temperature ranges. This avoids thermal emergencies and alleviates the need for thermal throttling. The overhead of vector-based scheduling is minimal; in case of multithreaded processors, vector-based scheduling even has negative overhead (increased throughput).

5 Resource-conscious Scheduling for Energy Efficiency

5.1 Introduction

The energy efficiency of a processor chip is influenced by its mode of operation (i.e., the operation frequency and voltage), the access patterns to resources like functional units, caches, or memory, and the contention for shared resources. The latter arises when a chip encompasses multiple execution contexts, as is the case with today's SMT and CMP processors. Resource utilization is a characteristic of the individual application, and the degree of contention for resources shared between the execution contexts depends on the applications running in parallel. Since it is the role of the scheduler to decide which applications to run at what time and in which combination, scheduling crucial for energy efficiency.

The design of most scheduling algorithms employed in today's operating systems stems from a time when multiprocessor systems were assembled of physically different single-core chips (traditional SMP systems). Hence, these schedulers do not take the peculiarities into account that arise when multiple threads are executed in parallel by one physical chip, as is the case with today's SMT and CMP hardware.

In case of an SMT chip, multiple logical threads of execution share the resources of one physical core. To a lesser extent, this is also the case for CMP chips, where memory access infrastructures and, in some cases, caches are shared by multiple cores residing on a chip. In most multiprocessor operating systems, scheduling decisions happen independently for each processor, that is, on every processor, a local scheduler decides which task to run at what time. Therefore, the combination of tasks running at a time is arbitrary, and situations in which several tasks utilizing the same resources are running in parallel are not precluded.

Contention for shared resources leads to stall cycles, i.e., pauses execution while the processor is waiting for a resource to become available. This reduces performance and, in addition, decreases energy efficiency, since power is dissipated without making progress.

Power management features such as frequency and voltage scaling create further interdependencies between the hardware threads of SMT and CMP chips, which need also be taken into account for achieving optimal performance and energy efficiency. On SMT chips, all logical processors share the frequency selected for the physical

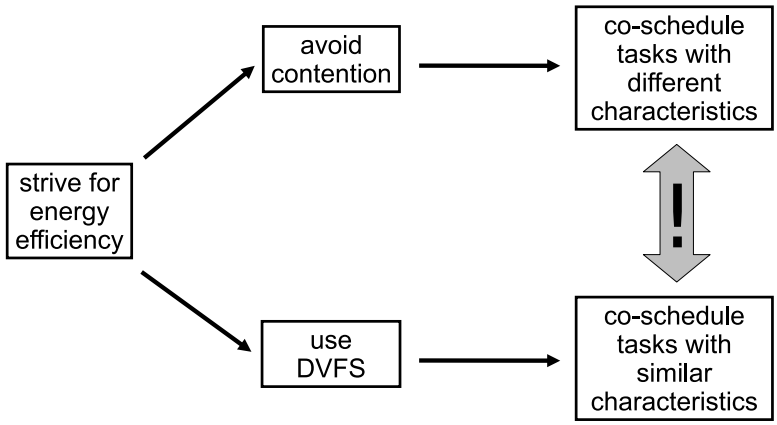


Figure 5.1: Two mutually exclusive ways for achieving energy efficiency

processor. In the case of CMP chips, it is possible to design hardware that supports setting a separate frequency and voltage for each core. However, commodity hardware often lacks this feature, since allowing multiple frequencies and especially multiple voltages at a time introduces additional hardware complexity [HM07, KGWB08].

The optimal frequency at which the processor can execute a task most efficiently in terms of runtime and energy depends on the task's characteristics, in particular on the frequency of memory accesses [WB02, CSP04, KDG+04]. The performance of memory-bound tasks depends on the speed of memory rather than on the processor speed; therefore, slowing down the processor by frequency scaling does not affect the runtime of such tasks as much as is the case for compute-bound tasks, whose performance is strongly correlated to processor speed. As a consequence, memory-bound tasks are most efficiently executed at lower processor frequencies and voltages, since a moderate increase in runtime and a reduced processor power consumption result in overall energy savings. Compute-bound tasks, on the other hand, are best executed at high frequencies, since the prolonged runtime introduced by lower frequencies would negate the power savings of frequency and voltage scaling.

If we consider both aspects, resource contention and frequency selection, the central question arises whether it is advantageous to run tasks with similar characteristics together in order to be able to run a SMT or CMP chip at the corresponding optimal frequency, or if tasks utilizing mutually different resources should be scheduled together in order to avoid contention (Figure 5.1). As an example, on a CMP, one possibility would be to schedule several memory-bound tasks at a time, which run most efficiently at a low frequency, and then to schedule several compute-bound tasks, which run most efficiently at a high frequency. On the other hand, with shared resources such as caches and memory interfaces, memory-bound tasks running together are likely to

suffer from resource contention [MM07], so it might be more advantageous to schedule memory-bound tasks together with compute-bound tasks.

Taking an Intel Core2 CMP chip as example, we analyze what is the optimal way to *co-schedule* tasks, i.e., which tasks should be selected to run in parallel on the execution contexts of the chip. In our analysis, we consider the criteria of resource contention and frequency selection. We find that in order to optimize the product of runtime and expended energy (energy delay product, EDP), the main goal must be to avoid contention by combining tasks with different characteristics; co-scheduling tasks that run best at a common optimum frequency does not pay off, since the decrease in energy efficiency introduced by resource contention cannot be offset by frequency scaling. Only if nothing but memory-bound tasks are available and contention cannot be avoided, it is beneficial to apply frequency scaling.

Activity vectors as introduced in Chapter 3 provide information about task characteristics. According to our analysis, we propose scheduling policies that make use of this information to co-schedule tasks with different characteristics in order to improve energy efficiency [MB08a].

The remainder of this chapter is structured as follows: In Section 5.2, we discuss related work in the area of contention-aware scheduling and in the area of optimal frequency selection. Section 5.3 presents our analysis of the effects of co-scheduling on resource contention and frequency selection. We describe the application of activity vectors to multicore scheduling in Section 5.4; the design of the scheduling policies is discussed in Section 5.5. To take advantage of frequency and voltage scaling, we propose to supplement our policies with a heuristic that lowers the frequency as a fallback solution when nothing but memory-bound tasks are running (Section 5.6). We implemented our proposed policies for Linux; Section 5.7 describes the implementation. The evaluation of our scheduling policies presented in Section 5.8 reveals that resource-conscious scheduling manages to reduce EDP considerably.

5.2 Related Work

Our work investigates the cross-effects of tasks running in parallel on one SMT or CMP chip. We address the problems of resource contention and optimal frequency selection with the goal of achieving energy efficiency by co-scheduling tasks based on their characteristics.

The concept and the term of co-scheduling have been introduced by Ousterhout [Ous82] several decades ago. The aim of co-scheduling is to schedule related threads of execution at the same time. Relation was originally defined by communication [Ous82, SW95] or synchronization [FR92] between threads. In the past, several approaches have considered co-scheduling to reduce contention for shared processor resources such as the functional units of SMT processors, or cache and memory bandwidth on multicore processors.

While there has been research dedicated to resource contention in SMP, SMT, and CMP systems as well as research in the area of selecting an optimal frequency for a specific workload, resource contention and frequency selection have hardly been considered in combination for improving energy efficiency.

Related work can be categorized into approaches addressing contention for shared resources of SMT chips, approaches addressing memory contention or cache contention in the context of SMP, SMT, and CMP systems, and approaches that explore the benefits of shared resources. Most approaches strive at reducing resource contention with the goal of improving performance, while the aspect of energy efficiency is not considered. Apart from this, a number of approaches are dedicated to the problem of frequency selection.

5.2.1 Contention for SMT resources

Snavely and Tullsen [ST00] combines tasks on a multithreaded processor in a way that results in maximum throughput, based on the rationale that the degree of resource contention depends on the combination of tasks executed in parallel. In contrast to our approach, the optimal combination of tasks is not inferred directly from the tasks' characteristics, but determined empirically by trying out different combinations.

Fedorova et al. [FSNS04] shows that the performance of SMT chips can be improved by co-scheduling high-IPC (instructions per cycle) tasks with low-IPC tasks. This policy is based on the idea that low-IPC tasks are often memory-bound and do not stress the other shared resources, while the opposite is the case for high-IPC tasks. However, as the authors state, IPC is not suitable to distinguish exactly which resources high-IPC tasks utilize. In addition, Fedorova et al. uses simulation instead of implementing the proposed policies in a real operating system.

Nakajima and Pallipadi [NP02] strives to minimize contention in SMT systems by evaluating performance monitoring counter readings, and in this respect constitutes the approach most closely related to vector-based scheduling. Nakajima and Pallipadi defines certain metrics that model the utilization of critical resources such as the integer unit, the floating point unit, and the L2 cache. However, these metrics are not seen with respect to a processor-specific maximum, as in our approach, but rather as an absolute number of accesses per time, which the proposed algorithm tries to balance across physical CPUs. Also, Nakajima and Pallipadi lets the kernel only provide the metrics and leaves the distribution of tasks to CPUs to userspace programs. This, however, does not allow to control the set of tasks executed in parallel if there are more tasks than CPUs, since in a time-sharing environment, the kernel-level scheduler determines the sequence of task execution on each CPU.

Parekh et al. [PELL00] and El-Moursy et al. [EMGAD06] characterize tasks using performance monitoring counters and, by simulation, study several scheduling algorithms that strive at improving the performance of an SMT processor. The approaches use metrics that directly or indirectly represent resource utilization and evaluate poli-

cies that avoid contention for shared resources by co-scheduling. The goal lies on optimizing performance-related metrics like IPC. In contrast to our approach, Parekh et al. proposes policies that improve performance at the expense of fairness by scheduling tasks that do not cause contention and yield the best performance, while penalizing unsuitable tasks by scheduling them less frequently. El-Moursy et al. favors a policy guided by the ratio between ready and in-flight instructions, a metric that is not obtainable using the performance monitoring counters deployed in today's processors.

Bulpin and Pratt [BP05] investigates the correlation between different performance counter metrics and task performance on an SMT processor. From the metrics, the most promising combination of tasks is inferred. The difference to our work is that the performance metrics are used to characterize combinations of tasks, not individual tasks. However, maintaining metrics for task combinations scales poorly with the total number of tasks, which can be a problem for large systems with many execution contexts and tasks.

McGregor et al. [MAN05] proposes scheduling based on resource utilization in multithreaded SMP systems. Similar to our proposed policy for avoiding contention, the approach pairs threads with complementary characteristics to run in parallel on a chip. In contrast to us, McGregor et al. uses a two-phase approach to enable co-scheduling: The first phase distributes tasks to physical processors in a way that balances memory bandwidth utilization, whereas the second phase distributes the tasks assigned to a physical processor to its logical processors based on either memory bandwidth, cache misses, or stall cycles. In addition, task characterization and scheduling is performed by a user-level processor manager instead of in the kernel.

5.2.2 Cache contention

Most of the research on avoiding resource contention on multicore chips has concentrated on cache as the constraining resource. Our experiments with the recent SPEC CPU 2006 benchmarks, however, suggest that contention for memory bandwidth is becoming more important than cache contention.

Siddha et. al. [SPM07] investigates the effects of placing tasks on cores sharing or not sharing L2 cache and shows that the performance impact depends on the degree of data sharing between the tasks and on the use each task can make of the cache.

Jiang et al. [JSCT08] analyzes the problem of distributing a set of tasks to a set of multicore chips and proves that the problem of finding an assignment of tasks to chips that yields minimal performance degradation is NP-complete for chips with more than two cores. Based on the assumption that cache is the single resource affected by contention, and that the performance degradation that results from co-scheduling a particular set of tasks is known in advance, the authors propose heuristic algorithms that come close to the optimal distribution. However, the approach relies on knowledge that typically is not available in real systems, and in addition does not consider the possibility of having more tasks than execution contexts.

Suh et al. [SDR02] proposes policies for co-scheduling tasks based on cache hit/miss counter information and on cache models. Simulation is used for evaluation, as the proposed policy would need additional counters not available in today's hardware.

Fedorova [Fed06] investigates scheduling policies that are aware of resource contention in CMP systems and aim at improving performance, fairness and predictability. The main difference to our approach is that L2 cache is considered the most critical resource, and that co-scheduling is deliberately not performed. Instead, the algorithms vary the number of concurrently executed tasks and the time quanta allotted to the individual tasks.

Chandra et al. [CGKS05] and Fedorova et al. [FSSN05] propose models for predicting the L2 cache miss rate that results from a combination of tasks, and suggest to use these models for co-scheduling tasks whose combination yields low miss-rates. However, the proposed models are computationally expensive, and obtaining the necessary input data for the models requires profiling of the tasks, which must either be done off-line or by running each task in isolation. This is prohibitive in a production system.

Banikazemi et al. [BPA08] uses the metrics of cycles per instruction, L1 and L2 cache miss ratios, L2 prefetch count, and floating point instruction count to characterize tasks. A user-level meta-scheduler employs a cache model to predict the impact of co-scheduling tasks onto cores sharing L2 cache based on the metrics, and sets the CPU affinity of the tasks accordingly.

Anderson and Calandrino [ACD06] proposes strategies for co-scheduling real-time tasks on a CMP that reduce contention for a shared L2 cache.

Zhao et al. [ZII+07] proposes to introduce new hardware monitoring features that provide the operating system with detailed information about the cache occupancy, cache interference, and cache sharing of tasks running on a CMP. The authors suggest to use this information for co-scheduling tasks with complementary demands for cache space or tasks that share common data in the cache, and demonstrate that this approach can yield significant performance improvements. Unfortunately, features like the ones proposed are not present in today's hardware.

5.2.3 Memory contention

Scheduling for reducing memory contention has been investigated in the context of multicore and traditional SMP systems. Moscibroda and Mutlu [MM07] investigates to what extent memory-intensive applications running in parallel on multicore processors slow down each other. Using an Intel Pentium D dual-core system, the authors show that an application's execution time can be increased by a factor of up to 2.9 if an application showing an inauspicious memory access pattern is running on a core of the same chip. The authors even argue that this interference could be abused for denial-of-service attacks.

Bellosa [Bel97b] identifies memory contention as a problem in multiprocessor real-time systems and proposes to throttle non-real-time tasks in order to guarantee memory bandwidth for real-time tasks.

Zhang et al. [ZDFS07] proposes to introduce performance metrics as a resource in operating systems and to use them for scheduling with the goal of avoiding resource contention. In particular, the approach evaluates using memory bus utilization as a metric to avoid memory bus contention in SMP systems and achieves significant improvements over resource-oblivious scheduling and scheduling guided by the metric of IPC.

Kondo et al. [KSN07] addresses memory bus contention in multicore processors. In contrast to our work, the approach assumes that separate frequencies can be chosen for the individual cores and tries to counteract the unfairness introduced by memory contention by slowing down cores that utilize too much memory bandwidth. Thus, frequency scaling is not used to improve energy efficiency, but to increase fairness. Similarly, Hedrich et al. [HII+09] proposes to throttle low-priority tasks in order to limit their interference with high-priority tasks via memory and cache contention. The authors suggest task characterization with performance monitoring counters as a possible way to infer whether there is resource contention between high and low-priority tasks.

Koukis and Koziris [KK06] investigates contention for memory and network bandwidth in the context of clusters of SMP systems. In particular, the authors devise a scheduling policy that selects ready-to-run tasks based on the bandwidth left over by already chosen tasks, which is similar to one of the policies we propose for avoiding contention. In contrast to our work, the authors propose a coarse-grained user-lever approach tailored to cluster systems. The approach is interesting in the respect that it also considers the contribution other devices than the CPU make to memory contention via DMA, which is beyond the scope of our work.

5.2.4 Profiting from shared resources

Some approaches aim at taking advantage of shared resources in order to improve performance or energy efficiency. Bellosa [Bel97a] proposes to take advantage of physically indexed caches that are shared between threads running in time-sharing fashion on a single core. The approach arranges runqueues in a way that threads accessing common data are scheduled in succession, so they can profit from the data that was loaded into the cache by their predecessors.

Thekkath and Eggers [TE94] investigates the potential of co-locating threads that access the same data onto SMT siblings in order to take advantage of the shared cache. The paper comes to the conclusion that no performance benefits are to be achieved this way, since for all applications studied, there is a high degree of locality for the shared data within each thread, and the overall number of invalidations and misses that occur when threads sharing data run on separate caches is low. Tam et al. [TAS07], on the

other hand, achieve speedups of 7% by clustering tasks that access common data on processors sharing cache using a SMP machine that features both SMT and CMP.

Rajagopalan et al. [RLA07] proposes a scheduling framework for *many-core* (i.e., large-scale CMP) processors that allows the user to guide the scheduling of application threads. The user specifies which threads are related, e.g., work on the same data or communicate frequently. With this knowledge, the scheduler places those threads on cores that share cache or are physically close to each other on the chip and thus have low communication overhead. In addition, the user can specify whether respecting data locality or balancing load over all cores is more important, and whether related threads should be co-scheduled.

De Vuyst et al. [VKT06] argues that on a chip that features both CMP and SMT, depending on the workload, it can be advantageous in terms of energy efficiency to schedule several tasks onto the logical processors of a subset of cores, while leaving other cores idle. This takes advantage of the fact that the logical processors of a core share the same physical resources and thus consume less power than logical processors of physically distinct cores. As a consequence, the authors propose policies for unbalanced scheduling. The approach is orthogonal to ours, since it assumes that there are fewer tasks than execution contexts, while our policies are suitable for scenarios with all contexts utilized.

On multicore processors with shared cache, variable portions of the cache can be allocated to the individual cores. This allows reducing the cache size for tasks that do not profit much from a larger cache either because of a small working set or bad data locality, and to increase the amount of cache allocated to tasks that do profit from a larger cache. Hsu et al. [HRIM06] investigates different policies for allocating cache to the cores of a CMP and finds that the optimal policy depends on the combination of applications that is running. While on today's CMPs, the cache allocation policy is implemented in hardware and fixed, Liu et al. [LSK04] proposes a shared L2 cache that is divided into splits. The splits can be assigned to individual cores, allowing the operating system to assign variable portions of cache to cores. Similarly, Rafique et al. [RLT06] proposes hardware mechanisms that allow the operating system to set quotas for cache utilization. This enables the operating system to allocate more cache to applications that profit most from it, in exchange for reducing the cache space allocated to applications whose performance does not significantly depend on cache.

5.2.5 Frequency selection

Previous research has investigated the problem of improving energy efficiency by selecting a frequency according to task characteristics in the context of embedded [WB02, CSP04, DR07, LCCF08], server [KGKR05], or cluster systems [HF05, FPL+07] as well as for notebook and desktop systems [HF04, SLSPH09]. Memory-bound tasks can be executed at lower CPU frequencies without significant slowdown, since memory throughput and not CPU speed is the determining factor for their perfor-

mance. In contrast, compute-bound tasks run more efficiently at higher frequencies, since lower frequencies prolong their runtime and cause them to consume power for a longer time, often negating the power savings gained by frequency scaling.

Many previous approaches that addressed frequency scaling for CMP systems were based on the assumption that a separate frequency can be chosen for each core (*per-core DVFS*, as opposed to *per-chip DVFS*).

Juang et al. [JWP+05] investigates per-core DVFS for multithreaded applications and shows that the decisions for setting the frequencies of the individual cores need to be coordinated in order to achieve an optimal EDP.

Isci et al. [IBC+06] investigates policies that aim at keeping a chip-wide power budged by using coordinated frequency scaling on chip multiprocessors. The approach assumes that the frequency can be scaled individually for each core and tries to optimize metrics like throughput or fairness while adhering to the power budget by considering the characteristics of the tasks running on each core.

Kong et al. [KCCC08] optimizes the EDP or ED^2 of applications running on a CMP chip by profiling the applications off-line and setting the optimal frequency independently on each core.

Rangan et al. [RWB09] assumes a chip whose cores run at different, but fixed frequencies. The authors investigate the potential of migrating hardware threads between cores of different frequencies instead of varying the frequencies of the cores. This avoids the overhead of frequency changes. Rangan et al. finds that the approach is superior to changing the cores' frequencies, but assumes that migrations are performed in hardware and on a timescale that is much smaller than operating system scheduling intervals. Similarly, Kotla et al. [KDG+04] assumes a CMP whose cores run at different frequencies. For deciding on which core (and hence at which frequency) to run a particular task, Kotla et al. develops a model based on memory intensity for predicting at which frequency a task runs most efficiently in terms of energy without losing too much performance.

Curtis-Maury et al. [CMSB+08] and Li and Martínez [LM06] investigate DVFS in combination with dynamic concurrency throttling (i.e., limiting the number of threads) for multithreaded applications on CMP systems. Curtis-Maury et al. also takes the assignment of threads to cores into account and therefore, like our work, combines DVFS with co-scheduling (albeit only considering setups with a smaller or equal number of threads than the number of cores). The approach assumes per-chip DVFS and uses event counters to characterize applications. In contrast to our approach, the events are selected automatically during an off-line calibration phase. At runtime, a predictive model is used to select the optimal setup (frequency, concurrency level, and assignment of threads to cores). Li and Martínez also assumes per-chip DVFS and uses a heuristic search to determine the most energy efficient setting concerning the number of active cores and the chip frequency that still meets a predefined performance target by empirically trying different configurations.

In the context of hardware design, the question has been addressed whether it is beneficial to support per-core DVFS instead of the more simple per-chip DVFS. Herbert and Marculescu [HM07] investigates the consequences of the restriction that several cores need to run at the same frequency and voltage, and comes to the conclusion that, for multithreaded workloads, the benefits gained by multiple frequency/voltage domains do not justify the additional complexity introduced for realizing these domains.

Kim et al. [KGWB08] investigates per-core DVFS using on-chip voltage regulators. Compared to the off-chip regulators employed today, this has the advantage of providing more frequency domains and quicker switching of frequencies at the cost of additional hardware complexity. The paper concludes that fast per-core DVFS can lead to energy savings, but must carefully be weighed against the hardware overhead it introduces.

Sharkey et al. [SBB07] compares several policies that aim at keeping a chip-wide power budget. The authors find global policies that coordinate the power management of the individual cores to be more efficient than independent local policies. Furthermore, they find that global policies that make use of per-core DVFS to set different frequency/voltage levels on the individual cores achieve only small improvements over policies that use per-chip DVFS.

5.3 Analysis of Resource Contention and Frequency Selection

Knowledge about the effects that different hardware and software mechanisms have on power consumption and performance is a prerequisite for designing energy-aware scheduling policies. In a first step, we analyze the effects of resource contention and frequency selection on the energy delay product (EDP), a metric that considers both energy efficiency and performance. In particular, we want to resolve the question whether it is preferable to co-schedule similar tasks in order to profit from DVFS, or whether tasks with different resource demands should be co-scheduled in order to avoid resource contention.

We use a CMP system as our test platform. In contrast to SMT, where almost all chip resources are shared, the cores of a CMP chip typically share only last level caches and memory access infrastructure. Thus, the latencies that occur upon cache or memory accesses lead to stall cycles during which no useful work is done. This is much more the case for CMP than for SMT, since for the latter, when a sibling is stalling on a memory or cache access, functional units can be used by the other siblings. As a consequence, frequency scaling as a means to reduce stall cycles is of particular importance for CMP chips.

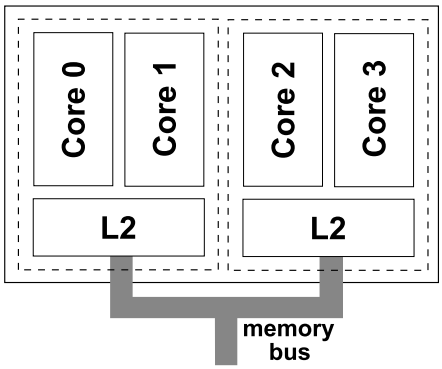


Figure 5.2: Architecture of the Intel Core2 Quad

Our findings are the following:

- The performance and the energy efficiency of a processor chip depend significantly on the combination of tasks running simultaneously on the cores. In particular, we find contention for memory bandwidth to be a crucial factor.
- Co-scheduling memory-bound tasks in order to be able to profit from frequency scaling is not beneficial, since the resulting memory contention diminishes energy efficiency to a degree that cannot be offset by frequency scaling.
- Instead, co-scheduling memory-bound with compute-bound tasks at the chip's maximum frequency results in an optimal EDP.
- Only if scheduling cannot avoid contention owing to a lack of compute-bound tasks, lowering the chip's frequency is indicated.

5.3.1 System description

As test platform for investigating the effects of resource contention and frequency selection, we chose a 2.4GHz Intel Core2 Quad Q6600, which is a quad-core CMP. The Core2 Quad Q6600 is a multi-chip module that consists of two silicon dies in one package. Each die comprises two cores sharing 4MiB of L2 cache (Figure 5.2). This allows observing the relevance of cache as a shared resource, since it is possible to conduct experiments on cores sharing or not sharing L2 cache.

The shared cache is allocated dynamically to the cores [Wec06]. This means cores can dynamically increase their utilization of the shared cache, based on the rationale that cores executing cache-intensive workloads profit from larger portions of the cache.

All four cores share a front side bus for accessing memory. Our test system uses a 266MHz front side bus and has 8GiB of DDR2 PC-6400 memory.

The processor supports scaling the frequency down to 1.6GHz. In this case, the core voltage is scaled from 1.24V to 1.13V. (For the rest of this chapter, when we speak of frequency scaling, we will imply that voltage scaling is also applied.)

The chip supports only one voltage setting for all four cores. It is, however, possible to run the two dies at different frequencies. Since scaling down the frequency of only one die does not allow to scale the voltage, this results only in marginal energy savings. Therefore, we will only consider two settings in our analysis: all four cores running at 2.4GHz and all four cores running at 1.6GHz. For all of the experiments described in this chapter, we ran the processor cores in 64-bit mode.

We also performed experiments with an AMD Opteron 2354 quad-core chip. In contrast to the Core2, the Opteron does not access memory via a front-side bus, but possesses an integrated memory-controller on the chip. The cores of the Opteron possess private L1 and L2 caches and all share a common L3 cache, which makes it harder to analyze the importance of cache contention than on the Core2, where two cores share a common cache, respectively.

Our basic finding is the same for both the Core2 and the Opteron—it does not pay off to co-schedule memory-bound tasks in order to be able to profit from lower chip frequencies. Therefore, and since the cache architecture of the Core2 allows a better analysis of cache contention, we will discuss our analysis of the Core2 in detail and only summarize the results for the Opteron at the end of this section.

5.3.2 Metric

The goal we want to achieve with the scheduling policies we propose in this chapter is to increase the energy efficiency of the processor. We choose the energy delay product as the metric we want to optimize. In many systems, the processor is the component requiring the most energy and cooling effort, so it is desirable to reduce its energy consumption. Energy efficiency is of paramount importance especially for multicore processors, since having multiple cores close together on a chip makes overall energy dissipation the main reason for thermal problems rather than localized heating, as is the case with SMT processors [YSBZ05]. On the other hand, it is undesirable to sacrifice too much performance. EDP considers both factors, energy efficiency and performance.

We deliberately consider only the energy consumption of the processor and not of other system components. This is a simplification, since we imply that the energy consumption of the other components is not influenced by scheduling, i.e., that the power consumption of all other components stays the same regardless which schedule is applied. Considering these components, especially memory that can be put into a low-power sleep state if not accessed [LFZE00, FEL03], is a topic for future work.

Since we perform our measurements on real commodity hardware, we are only able to determine the energy consumed by the processor chip in total, but are not able to distinguish how much energy was consumed by each individual core. We can therefore only give EDP as a *per-chip* metric (runtime of a task multiplied by the energy consumed by the entire chip, divided by the number of cores), attributing equal amounts of power to the individual cores, and not as a *per-core* metric (runtime of the task multiplied by the energy consumed by the core the task ran on).

Prior research on energy efficiency for CMPs, for example Juang et al. [JWP+05], Isci et al. [IBC+06], or Herbert and Marculescu [HM07], has used simulation and power estimation, which allows to determine the amount of energy each simulated core consumes. Measuring per-core energy in a real physical system, however, is only possible if the cores are supplied with power via separate lines, which, is typically not the case. Also, the power consumption of components shared by multiple cores, such as shared caches or the memory interface, can, in practice, not be attributed to a task running on a specific core, which makes per-chip EDP the only sensible metric for our study.

The per-chip EDP of a task is always dependent on the tasks running on the other cores, and EDP values must therefore always be seen with respect to the specific workload. We believe that chip-specific EDPs are a valuable metrics for evaluating the benefits of our policies if we consider the following guidelines:

- Since the EDP of a task depends on the tasks running on the other cores and thus on the workload, we only compare EDPs of tasks within the same workload. For example, we compare the EDPs of a certain combination of tasks under vector-based scheduling to the EDPs of the same combination of tasks under standard Linux scheduling. This way, even if the actual distribution of power consumption to cores differs from the assumption that all cores consume equal amounts of power, the overall trend (EDPs increase or EDPs decrease) is still the same for the core-specific and the chip-specific EDP metric.

- When comparing the efficiency of our policies for different workloads, we do not compare individual tasks, but always use the average EDP of all tasks of a certain workload.

- Comparison of EDPs from tasks within a workload only reveals the effects of the policy on the relative runtime of the tasks. It does not capture the change in power consumption caused by the policy, since all EDP calculations are based on average power. For instance, if a certain policy would cause the power consumption of only one particular task do drop, this would result in a lower per-chip-EDP of all tasks in the workload. However, since we do not consider applying frequency scaling to individual cores, but only chip-wide, it is unlikely that a change in policy has a big effect on the relative power consumptions of

the tasks: The relative reduction in power achieved by frequency scaling is approximately the same for different types of applications, as we verified with experiments using SPEC CPU 2006.

- In addition, we only evaluate the energy efficiency for scenarios with all cores occupied. This avoids the problem of attributing the power consumed by idle cores.

5.3.3 Energy measurements

For performing energy measurements, we instrumented the power supply of the processor chip. We inserted a 5mΩ resistor into the 12V lines running from the power supply to the motherboard. This allows us to determine the power consumption of the processor including the voltage regulators, which are supplied via the 12V lines.

We used a National Instruments SC-2345 board connected to a measurement computer to sample the voltage drop at the resistor at a granularity of 10,000 samples per second. This enables us to infer the power delivered via the 12V line, and in turn, to obtain the energy consumed by the processor during a given period of time.

5.3.4 Resource contention

Methodology

In this section, we evaluate the problem of resource contention in isolation, i.e., at a fixed chip frequency. Unless stated otherwise, the experiments presented in the following subsections were performed at the chip's maximum frequency of 2.4GHz.

In Section 5.3.5, we evaluate the aspects that dynamic frequency scaling has on different types of workloads. Finally, Section 5.3.6 brings together both aspects, resource contention and frequency selection.

Microbenchmarks

First, we evaluate resource contention between the cores using several microbenchmarks. The resources the cores are contending for are L2 cache (shared by two cores, respectively) and memory bandwidth (shared by all four cores).

We select microbenchmarks that differ in their use of the named resources. `aluadd` is a compute-bound microbenchmark compiled from hand-written assembler code that performs integer additions exclusively on the CPU's registers. Hence, this microbenchmark does not utilize any resources shared between cores. As a memory-bound microbenchmark, we use the `stream` memory benchmark [McC95], which causes heavy utilization of the shared memory bus. We also create modified cache-bound versions of the benchmark by sizing the working set to fit into the shared L2 cache once (`stream-fit1`) or twice (`stream-fit2`). Since the microbenchmarks

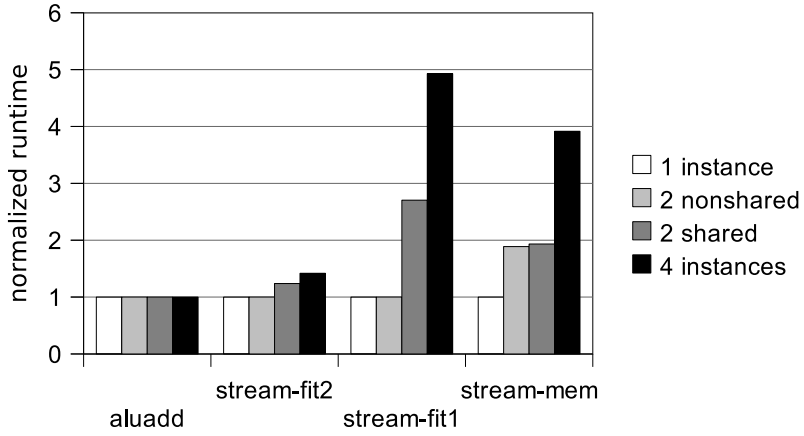


Figure 5.3: Normalized runtime of microbenchmarks

possess only relatively short runtimes in the order of seconds, in this experiment and all further experiments involving microbenchmarks, we run the benchmarks in loops and report the average runtimes.

Figure 5.3 shows the runtimes of the microbenchmarks when running alone, together with another instance of the same benchmark running on a core using a different L2 cache, together with a instance on a core using the same cache, and together with three instances on the other cores. All runtimes are normalized to the runtime of an instance running alone.

As expected, `aluadd`'s runtime is not influenced by instances running on other cores. `stream-fit2`'s runtime increases slightly when another instance uses the same cache. Although the cache's capacity is big enough for holding both task's working sets, conflict misses, that is, too many addresses being mapped into the same cache set, can still occur. `stream-fit1`'s runtime increases considerably when two instances share a cache. Since now, the combined working sets of the benchmarks do not fit into the cache, a lot of capacity misses occur, necessitating frequent memory accesses and causing slowdown. When four instances are running, memory contention causes a further increase in runtime. Finally, the original memory-bound `stream` benchmark suffers from memory contention already when two instances are running on different caches.

SPEC CPU 2006

We performed the same evaluation using the SPEC CPU 2006 benchmarks. Figure 5.4 depicts the runtimes of the benchmarks, sorted by the slowdown four instances suffer

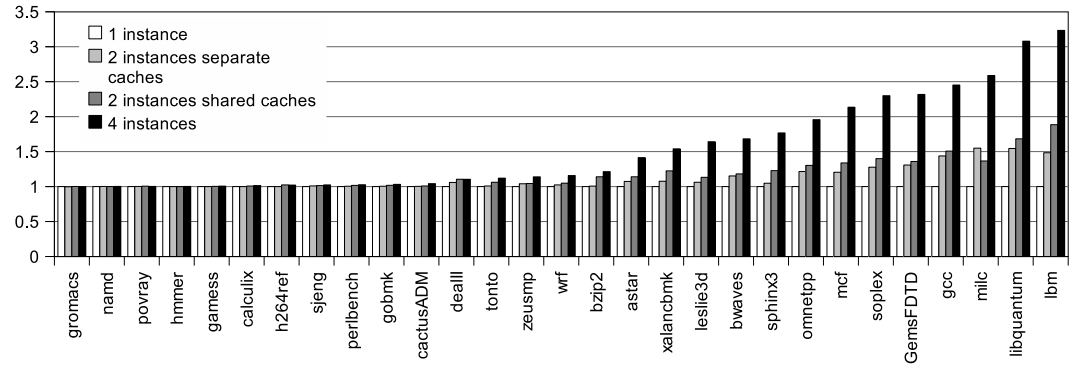


Figure 5.4: Normalized runtime of SPEC benchmarks

in comparison to one instance running alone. Many SPEC benchmarks (those on the left side of the figure) behave like `aluadd` and `stream-fit2`, showing no or only little slowdown even when combined on the same cache, which indicates that these benchmarks hardly suffer from resource contention. The rest of the benchmarks (on the right side) all show a considerable increase in runtime when multiple instances are running. It is remarkable that the benchmarks that do suffer substantially reduced performance with multiple instances do so already when two instances are running on cores with separate caches, which indicates that memory contention is responsible for the slowdown. The most affected benchmark is `lbm`, which shows a slowdown of factor 3.2.

The `milc` benchmark is a special case since it runs faster on shared caches than on separate caches. We performed further experiments with the benchmark and noticed that the anomaly disappeared when we disabled the speculative prefetching of data from memory to the cache the Core2 performs by default. In addition, the runtime of `milc` decreases when prefetching is disabled. This indicates that `milc` does not profit from prefetching. On the other hand, prefetching useless data causes additional memory contention. When running with separate caches, prefetching is done for two caches instead of just one, as is the case with a single shared cache. This explains the anomaly.

Importance of memory contention

Our experiments indicate that memory contention is a problem for many SPEC benchmarks, and that the case that only cache is the constraining resource is rare. As mentioned, the SPEC benchmarks either suffer from no or only little resource contention, or suffer from (memory) contention already when two instances run on cores not sharing cache. Behavior like that of the microbenchmark `stream-fit1` (slowdown when

running on shared, but not when running on separate caches, indicating that cache is the limiting factor) is only observed for `bzip2`, `xalanbmk` and `sphinx3`; however, the overall slowdown for two instances of these benchmarks is not nearly as severe as for `stream-fit1`. Therefore, and since cache contention has already been addressed in numerous previous studies (e.g., Suh et al. [SDR02], Chandra et al. [CGKS05], Fedorova [Fed06], Siddha et al. [SPM07]), we will concentrate on memory bandwidth as constraining resource.

Research on contention in CMP systems that has concentrated on cache as the constraining resource has mostly used the older SPEC CPU 2000 benchmark suite for evaluation. An analysis of the SPEC CPU 2000 benchmarks by Lee [Lee06] reveals that only few of these benchmarks are memory-bound. Bird et al. [BPJ+07] compares SPEC CPU 2006 to its predecessor SPEC CPU 2000 and finds SPEC CPU 2006 to cause considerably more L2 cache misses and thus to stress the memory subsystem considerably more than SPEC CPU 2000.

According to the SPEC documentation, the memory space required to run the SPEC benchmarks is four times higher for SPEC CPU 2006 than for SPEC CPU 2000 (1GB vs. 250MB). When compiling SPEC CPU 2006 for a 64-bit architecture, even 2GB are recommended. This indicates that demands for memory bandwidth have also increased between the SPEC versions. The fact that SPEC is intended to be representative for real user application workloads suggests that these applications also tend to become more dependent on memory bandwidth.

Interaction of different benchmarks

Up to now, we have investigated resource contention between instances of the same benchmark. To investigate how instances of different benchmarks affect each other, we selected three memory-bound benchmarks heavily affected by resource contention in the previous test (`lbm`, `libquantum`, and `mcf`) and two compute-bound benchmarks hardly affected by resource contention (`hmmer` and `gromacs`). We ran every possible combination of the five selected benchmarks on two cores with separate L2 caches. (We also performed the experiment with cores sharing L2 cache and obtained similar results.) Since the benchmarks have different runtimes, in order to have a constant load, we re-started all benchmarks with shorter runtimes until the benchmark with the longest runtime terminated, but only considered the data of the first run for each benchmark.

Figure 5.5 shows the runtimes of the benchmarks, normalized to the time it takes to run the respective benchmark alone. As expected, benchmarks that were affected by resource contention in the previous test also suffer from resource contention when running in combination, whereas benchmarks that were not affected by resource contention in the previous test do not lead to resource contention when combined.

When one benchmark from the first and one from the second group are combined, neither of the benchmarks is slowed down significantly. This demonstrates the poten-

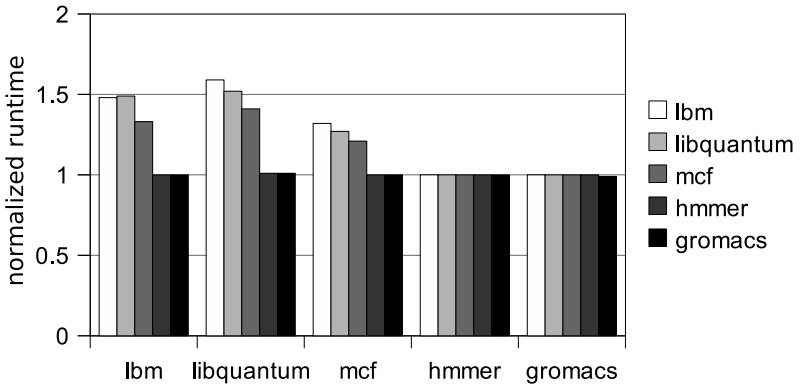


Figure 5.5: Combinations of memory-bound and compute-bound benchmarks

tial for reducing the runtime (and thus also the energy consumption) of memory-bound tasks by running them in combination with compute-bound tasks.

Effects on energy efficiency and EDP

Resource contention causes performance degradation and prolonged task runtimes. The increase in runtime influences EDP in two respects. Firstly, the increased runtime directly leads to an increase of EDP, since EDP is the product of runtime and energy. Secondly, resource contention causes tasks to consume power for a longer time, which increases the amount of energy required to run a task: As explained in Chapter 4, the power consumption of a processor can be divided into a dynamic part that depends on switching activity, and a static part independent from activity. With resource contention, activity is stretched out over a longer period of time. Thus, the static power, which is independent from activity, is consumed for a longer time and results in increased energy requirements. This leads to a further increase of EDP.

To quantify the implications of resource contention on EDP exactly, we would have to measure the energy consumption of a task susceptible to resource contention once with and once without the presence of contention. This, however, would require changing the workload, (e.g., combining the task concerned once with memory-bound and once with compute-bound tasks), because contention always depends on the combination of tasks running. Since we can only measure the power consumption of the chip as a whole, this would lead to incomparable results, as the power consumption depends on *all* tasks running on the chip. (We can determine EDP only as a workload-dependent per-chip metric; cf. Section 5.3.2.)

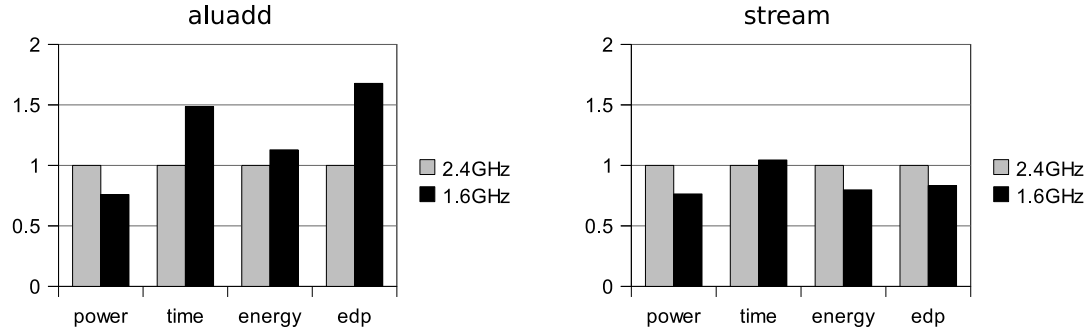


Figure 5.6: Effects of frequency scaling on aluadd and stream

Yet, we can make the qualitative statement that resource contention leads to a dramatic increase of EDP. As we have seen, contention leads to a huge increase of runtime, and because of the additional increase in energy consumption, EDP is increased even more. Thus, from the standpoint of EDP like from the standpoint of performance, it is beneficial to co-schedule memory-bound and compute-bound tasks.

5.3.5 Frequency selection

Microbenchmarks

For investigating the problem of frequency selection, i.e., choosing the best frequency for a given workload, as a preliminary test, we ran different combinations of the `aluadd` and the `stream` benchmark on the cores.

Figure 5.6 shows the effects of frequency scaling on the two microbenchmarks. The figure depicts power consumption, runtime, energy, and EDP of the benchmarks at 2.4GHz and at 1.6GHz, normalized to the respective values at the full frequency of 2.4GHz. The data for the figure was sampled for a combination of four instances of the respective microbenchmark.

Since `aluadd` is compute-bound and thus depends purely on the speed of the CPU, its runtime increases when the frequency is scaled. This increase outweighs the decrease in power consumption achieved by frequency scaling, so the energy, and even more the EDP, increase at the lower frequency.

`stream`, on the other hand, is memory-bound and thus depends largely on the available memory bandwidth, which is independent from the CPU's frequency. Accordingly, the runtime of `stream` hardly increases when the CPU frequency is lowered. As a result, the energy required to run the benchmark—and as a consequence, the EDP—decreases because of the reduced power consumption at lower frequencies. It

instances	aluadd time	aluadd energy	aluadd EDP	stream time	stream energy	stream EDP	average EDP
4 aluadd	1.49	1.16	1.68	—	—	—	1.68
3 aluadd + 1 stream	1.49	1.08	1.63	1.13	0.83	0.93	1.45
2 aluadd + 2 stream	1.49	1.10	1.60	1.07	0.77	0.82	1.23
1 aluadd + 3 stream	1.49	1.10	1.60	1.09	0.85	0.93	1.13
4 stream	—	—	—	1.04	0.80	0.83	0.83

Table 5.1: Relative runtime, energy, and EDP of benchmark combinations

is notable that, at 1.6GHz, the increase in EDP for `aluadd` is much bigger than the decrease for `stream`, which suggests that when combinations of the two microbenchmarks are running, frequency scaling is not beneficial.

Table 5.1 shows the results of heterogeneous combinations of microbenchmarks running together. For each microbenchmark, the table shows the factor by which runtime, energy and EDP of the microbenchmark change when the frequency is reduced from 2.4GHz to 1.6GHz. The last column shows the average change of EDP.

When looking at the average, as expected, only a combination of four memory-bound tasks justifies frequency scaling; that is, lowering the frequency is only beneficial if tasks suffering from memory contention are running on all cores. As suggested earlier, the reason is that the increase of EDP, which frequency scaling introduces for compute-bound tasks, is much higher than the reduction of EDP achieved for memory-bound tasks. In the following, we show that the same holds true for more realistic workloads—the SPEC CPU 2006 benchmarks.

SPEC CPU 2006

Figure 5.7 depicts runtime, energy and EDP for the SPEC benchmarks when run at 1.6GHz, relative to the respective values at 2.4GHz. All values were obtained by running four instances of a benchmark simultaneously on all cores. The order in which the benchmarks are depicted is the same as in Figure 5.4. The figure shows that the benchmarks that do profit from frequency scaling (i.e., show a lower EDP at 1.6GHz) are all located on the right side, which means they are memory-bound (cf. Section 5.3.4) and thus susceptible to resource contention. The compute-bound benchmarks on the left, in contrast, all show considerably higher EDPs at 1.6GHz compared to 2.4GHz, since, like `aluadd`, their runtime increases considerably with the lower frequency.

To show that frequency scaling is not only beneficial for homogeneous workloads where all cores are running instances of the same benchmark, we ran combinations

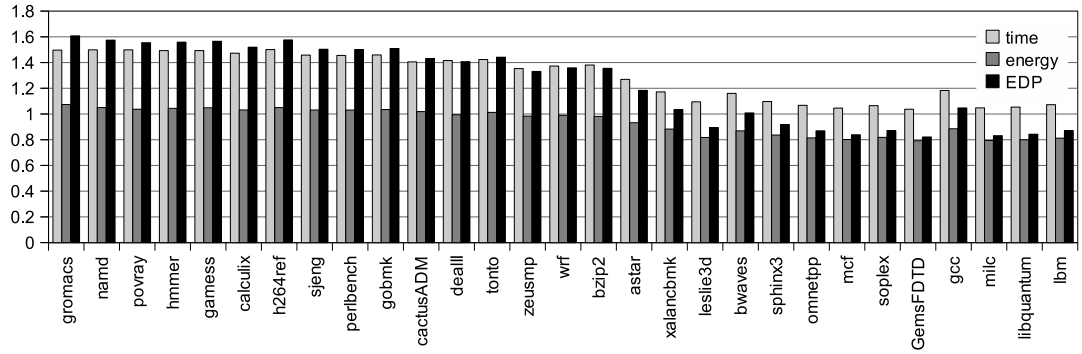


Figure 5.7: Relative runtime, energy, and EDP for the SPEC benchmarks

core1	core2	core3	core4	time	energy	EDP
libquantum	omnetpp	libquantum	omnetpp	1.06	0.8	0.85
libquantum	mcf	libquantum	mcf	1.05	0.79	0.83
omnetpp	mcf	omnetpp	mcf	1.06	0.8	0.85
libquantum	mcf	omnetpp	mcf	1.05	0.79	0.83
libquantum	omnetpp	omnetpp	mcf	1.05	0.8	0.86
libquantum	omnetpp	libquantum	mcf	1.08	0.82	0.89

Table 5.2: Combinations of memory-bound benchmarks—relative runtime, energy, and EDP at 1.6GHz compared to 2.4GHz, averaged over the benchmarks involved

of different benchmarks that showed a reduced EDP at 1.6GHz. Table 5.2 shows the runtime, energy, and EDP sampled at 1.6GHz relative to the respective metric at 2.4GHz. The values are averaged over all benchmarks of the respective scenario. For all combinations, the effect of attaining a reduced EDP at 1.6GHz persists.

We repeated the test, but combined benchmarks that showed a reduced EDP at 1.6GHz (`mcf`, factor 0.84), an increased EDP (`hmmer`, factor 1.56) and roughly the same EDP (`bwaves`, factor 1.01) (Table 5.3). In particular, we can observe that, on average, all combinations that include the compute-bound `hmmer` show a worse EDP at 1.6GHz. This is even the case for one instance of `hmmer` combined with three instances of the memory-bound `mcf`. An improved EDP at 1.6GHz is only observable for two or three instances of `mcf` combined with the moderately memory-bound `bwaves`.

This confirms that the results we achieved with the combinations of microbenchmarks persist for real applications like the SPEC benchmarks: If memory-bound and

core1	core2	core3	core4	time	energy	EDP
hmmer	bwaves	hmmer	bwaves	1.37	0.98	1.35
hmmer	bwaves	hmmer	mcf	1.36	0.98	1.34
hmmer	mcf	hmmec	mcf	1.34	0.95	1.31
hmmer	bwaves	bwaves	bwaves	1.28	0.93	1.2
hmmer	bwaves	bwaves	mcf	1.28	0.92	1.2
hmmer	mcf	bwaves	mcf	1.22	0.88	1.09
hmmer	mcf	mcf	mcf	1.16	0.87	1.04
bwaves	bwaves	bwaves	mcf	1.19	0.87	1.04
bwaves	mcf	bwaves	mcf	1.13	0.82	0.93
bwaves	mcf	mcf	mcf	1.05	0.77	0.83

Table 5.3: Combinations of memory-bound and compute-bound benchmarks—relative runtime, energy, and EDP at 1.6GHz compared to 2.4GHz, averaged over the benchmarks involved

compute-bound tasks are running in combination, the EDP is best at high frequencies; only four memory-bound tasks running in parallel justify lowering the frequency.

Intermediate frequencies

Besides the two documented operation points of 2.4GHz/1.24V and 1.6GHz/1.13V, the Core2 Q6600 can also be operated at 1.82GHz and at 2.13GHz by manually setting the frequency multiplier. We also ran the SPEC benchmarks at these two frequencies (with corresponding voltages of 1.16V and 1.20V, respectively). However, the additional operation points yield only little benefits: All benchmarks except four are executed most efficiently in terms of EDP at either the maximum frequency (compute-bound benchmarks) or the minimum frequency (memory-bound benchmarks), with efficiency degrading monotonously towards the opposite end of the frequency spectrum.

The four exceptions are `bwaves`, `gcc`, `sphinx3` (optimal frequency of 1.82GHz), and `xalanbmk` (optimal frequency of 2.13GHz), which are exactly the benchmarks that lie somewhere in between compute-bound and memory bound (cf. Figures 5.4 and 5.7). Out of these four benchmarks, only for `gcc`, there is a significant improvement of EDP at the optimal frequency (10%), while the other three benchmarks only yield minor improvements compared to the maximum or minimum frequency (1% for `bwaves` and `sphinx3`, 3% for `xalanbmk`). For these reasons, we do not consider the intermediate frequencies in our further experiments.

5.3.6 Optimal co-scheduling

The experiments described in the preceding sections indicate that (1) co-scheduling memory-bound and compute-bound tasks reduces contention and, as a consequence, EDP and (2) that frequency scaling reduces EDP if only memory-bound tasks, but not if memory-bound and compute bound tasks are co-scheduled.

On a multicore chip, if there are more tasks eligible for execution than there are cores, the question arises whether it is better to run memory-bound tasks together in order to be able to profit from frequency scaling, or to run compute-bound with memory-bound tasks in order to avoid resource contention. In a system consisting of several multicore chips, a similar question arises regarding the distribution of tasks to chips.

The experiments in Section 5.3.4 indicate a huge performance penalty for memory-bound tasks running simultaneously (see Figures 5.3 and 5.4). The `stream` microbenchmark is slowed down by a factor of 3.9, the SPEC benchmark `lbm` by a factor of 3.2 when four instances of the respective benchmark are running in parallel. As a consequence, there is a significant increase in energy consumption. On the other side, frequency scaling achieves a reduction of energy consumption by a factor of 0.77 at best (c.f. Tables 5.1 and 5.2). Thus, the slowdown introduced by contention outweighs the reduction of power consumption achievable with frequency scaling by far. Since a longer runtime means consuming power for a longer time, combining compute-bound tasks and engaging frequency scaling yields an overall increase in energy consumption despite the lower power consumption. The effect is even bigger for EDP, which considers both, energy and runtime.

When looking at Figure 5.7, the least memory-intensive SPEC benchmark that profits from frequency scaling is `leslie3d`. However, `leslie3d` already suffers a slowdown of factor 1.64 when four instances are co-scheduled (cf. Figure 5.4). Thus, in summary, the EDP is worse for memory-bound tasks running in combination, even when frequency scaling is applied, than for memory-bound tasks running together with compute-bound tasks at the highest frequency.

We want to illustrate this with the results of an experiment with two SPEC benchmarks. In our scenario, we want to run four instances of `soplex`, a typical memory-bound benchmark we found to profit from frequency scaling, and four instances of `hmmer`, a completely compute-bound benchmark, on our quad core. We compare the following three scheduling scenarios:

1. Always run the four instances of `hmmer` at their optimal frequency of 2.4GHz, then run the four instances of `soplex` at their optimal frequency of 1.6GHz.

2. Always run two instances of `hmmer` with two instances of `soplex` at a time at 2.4GHz

3. Always run two instances of `hmmer` with two instances of `soplex` at a time at 1.6GHz

	time [s]		energy [KJ]		EDP [MJs]		
Scenario	hmmer	soplex	hmmer	soplex	hmmer	soplex	average
1: hmmer @ 2.4GHz, soplex @ 1.6GHz	923	1310	17.0	13.7	15.6	18.0	16.8
2: hmmer + soplex @ 2.4GHz	952	837	15.7	13.8	15.0	11.6	13.3
3: hmmer + soplex @ 1.6GHz	1420	911	17.0	10.9	24.1	9.9	17.1

Table 5.4: Impact of different schedules and frequency settings

Table 5.4 shows the runtime, CPU energy, and EDP for the scenarios. For scenario 1, resource contention slows down the four instances of `soplex` running in parallel, negating the power savings that result from frequency scaling; the energy consumption of `soplex` is as high as in scenario 2, where `soplex` runs at 2.4GHz. In scenario 3, running `hmmer` at 1.6GHz increases its runtime substantially compared to the other scenarios. The high runtimes of `soplex` in scenario 1 and of `hmmer` in scenario 3 clearly favor scenario 2. As expected, scenario 2 shows the best average EDP; here the benchmarks can be executed requiring only 80% of the EDP shown by the other two scenarios.

Motivated by the results of this analysis, our scheduling policies presented in Section 5.5 strive to combine tasks with different characteristics, and only engage frequency scaling as a fallback if nothing but memory-bound tasks are available.

5.3.7 Results for the AMD Opteron

We also conducted experiments with the microbenchmarks well as with the SPEC benchmarks on a 2.2GHz AMD Opteron 2354 quad-core. As on the Intel Core2, memory-bound benchmarks scheduled together suffer from substantial slowdown because of contention, although the slowdown is less severe than with the Core2. We attribute this to a better performance of the integrated memory controller in the Opteron, as opposed to the front-side bus used in the Core2. Four instances of `stream` running together on the four cores of the Opteron are slowed down by a factor of 2.7 compared to a single instance (Core2: factor 3.9); the most memory intensive SPEC benchmark, `lbm`, suffers a slowdown of factor 2.5 (Core2: factor 3.2).

Our test chip supports frequency scaling to 2.0, 1.7, 1.4, and 1.1GHz. Voltage is scaled accordingly, but all four cores are required to run at the same voltage. As with the Core2, the benchmarks that profit from frequency scaling are exactly the ones that are affected most by memory contention. Since the Opteron allows lower frequencies than the Core2, it offers more potential for conserving energy for memory-bound tasks. `lbm` profits most from frequency scaling; at the lowest processor frequency of 1.1GHz, the benchmark can be executed with only 0.64 times the EDP compared to 2.2GHz. (Core2: factor 0.87 at 1.6GHz).

Although memory contention is not as severe on the Opteron as on the Core2 and although the Opteron can improve energy efficiency for memory-bound tasks more than the Core2, this still does not warrant co-scheduling memory-bound tasks. Like with the Core2, alone the increase in runtime that results from co-scheduling memory-bound tasks offsets the reduction of EDP that is achievable by frequency scaling, as can be seen at the example of `lbm`. The same is the case for the other memory-bound benchmarks, which suffer from less slowdown when co-scheduled, but for which frequency scaling also yields smaller savings.

5.4 Activity Vectors for Multicore Scheduling

Before we introduce the design of our scheduling policies, we want to discuss how we apply the concept of activity vectors for characterizing tasks as a base for resource-conscious scheduling.

In Chapter 3, we have mentioned that the choice of components represented by an activity vector depends on the characteristics of the platform. For SMT processors, many resources such as ALUs, FPUs, branch predictors, reorder buffers, and caches are shared between siblings. Thus, for contention-aware scheduling on those processors, activity vectors with many components representing the individual chip units make sense. An example are the activity vectors on the Intel NetBurst architecture we introduced in Chapter 4.

For multicore processors, we choose to use the concept of activity vectors at a coarser granularity than for SMT processors. The reason is that contention on multicore chips affects just two resources: memory bandwidth and cache. In addition, we include a third component in the activity vector that represents all non-shared resources for which there is no contention.

Representing the utilization of non-shared resources in the activity vector is useful since otherwise, tasks that do not utilize shared resources would possess a zero vector as activity vector. This would be problematic for scheduling policies that are based on angle calculations (cf. Section 4.4). In addition, the utilization of non-shared resources is typically lower for tasks that are slowed down by contention for other resources and thus serves as further means for distinguishing between tasks that suffer or do not suffer from contention. However, since the utilization of the individual resources for which there is no contention is insignificant for interference between the tasks, it is appropriate to accumulate the utilization of these resources in one vector component.

Hence, the three resources considered by the activity vector are: memory bandwidth, L2 cache, and non-shared. While memory bandwidth and L2 cache are the resources for which there is contention, the resource "non-shared" stands for all resources that are not shared between cores, such as, for instance, L1 cache or integer and floating point units.

As discussed in Chapter 3, we define the utilization of all resources by access frequency. For memory, bandwidth, or in other words, the number of accesses that the cores can make per time, is limited by the memory interconnect and the speed at which the memory chips can deliver data. Therefore, it is sensible to define memory utilization as the number of memory accesses initiated by a core during a certain period of time divided by the maximum number of accesses the system supports during that period of time. Since the cores of a chip share the memory interconnect, multiple cores showing high memory utilization indicates contention.

In contrast, for the L2 cache, bandwidth is not a problem, since typically a multi-port, multi-bank cache [SF91] and a crossbar interconnection is used, allowing multiple cores to access the cache simultaneously [KZT05, HKS+07]. Our experiments with the microbenchmarks in Section 5.3.4 reflect this: The microbenchmark `stream-fit2`, whose working set fits into the cache twice, shows little interference if two instances have to share a cache, although both instances heavily stress the L2 cache.

However, the available space in the cache is a limited resource. Therefore, it would be natural to define cache utilization by the percentage of cache lines a task occupies, as proposed, for instance, by Zhao et al. [ZII+07]. Unfortunately, such information is not easily obtained, since today's commodity hardware does not export information about cache allocation to the software.

On the other hand, cores that perform more cache accesses have the potential of evicting other cores' cache lines more frequently and thus of occupying greater portions of the shared cache. Since access frequency can be obtained more easily than space occupancy, we use the access frequency for determining cache utilization and define cache utilization, like memory bandwidth utilization, as the number of cache accesses a core initiates, divided by the maximum number possible. As with other resources, the maximum can either be inferred from the hardware specification, or be determined empirically via microbenchmarks.

5.4.1 Frequency dependency of activity vectors

For maximum energy efficiency, we will run different tasks at different frequencies. The frequency at which a task is running has an impact on the task's resource utilization, and thus the task-specific utilization of a particular chip resource is likely to change upon a frequency switch. In particular, the maximum number of accesses a resource supports during a period of time can, depending on the resource, be dependent on the chip's frequency.

Since activity vectors represent a task's resource utilization, we have to consider the effects of different chip frequencies with respect to task activity vectors. We accomplish this by introducing the concept of *translation vectors*. A translation vector allows to infer how the resource utilization of a task running at a certain frequency can be expected to change when the task is running at another frequency.

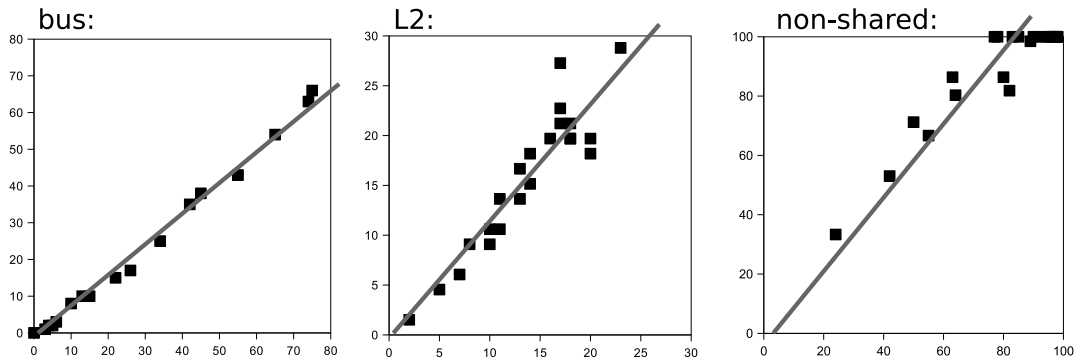


Figure 5.8: Resource utilization caused by the SPEC benchmarks at 2.4GHz (x-Axis) compared to 1.6GHz (y-Axis) in percent

Frequency changes affect the utilization of different resources in different ways. The reason is that the memory and memory interconnect usually operate with a clock separate from that of the chip, and in most of today's systems, frequency scaling only affects the processor clock, while memory keeps operating at the same speed as before.

Therefore, the utilization of memory bandwidth effectively decreases when frequency scaling is engaged: At a lower frequency, the processor can issue fewer memory requests per time, since the calculations done between two successive memory accesses take longer.

On the other hand, if the chip is running at a reduced frequency, memory is faster in relation to the chip. Thus, the utilization of chip resources such as the ALUs increases, since the number of cycles during which these resources are idle while waiting for memory decreases. The same holds true for the L2 cache, whose frequency is scaled down with the chip.

The dependency of resource utilization from frequency becomes apparent when looking at the three resources represented by activity vectors for a specific application run at different frequencies. Figure 5.8 shows the dependency of the utilization of the memory bandwidth, the L2 cache, and the non-shared chip components on frequency for our test chip. We ran all SPEC CPU benchmarks once at 1.6GHz and once at 2.4GHz, and monitored the utilization of the named resources for each benchmark. In the diagrams, we use the average utilization at 2.4GHz as the x–value and the average utilization at 1.6GHz as the y–value. Each point represents one benchmark, whereas the straight line denotes the trend.

The frequency dependency of resource utilization raises the question of how to define task activity vectors with respect to chip frequency. The options are defining an activity vector as the actual resource utilization of a task at the frequency the task

is currently running at, or as the resource utilization the task would cause at a certain fixed reference frequency, regardless of the actual frequency the task is running at.

A consequence of the first option would be that we could not simply compare activity vectors of tasks that are running at different frequencies. For instance, assume two tasks T_1 and T_2 running on a core. The core first executes T_1. Then, T_2 is scheduled and, according to some policy, the frequency of the chip is reduced. Now assume that while T_1 was running, the memory bandwidth utilization was higher than while T_2 was running. If we simply define the activity vectors of the tasks as the resource utilization they caused while running, the component of the activity vector representing memory bandwidth utilization would be higher for T_1 than for T_2. If both tasks were executed at the same frequency, however, it could well be that T_2 was more memory intensive than T_1, and that the lower memory utilization in our scenario was only caused by the low frequency T_2 was executed at in comparison to the frequency T_1 was executed at. Thus, if we defined activity vectors as the actual utilization a task causes at the frequency it is currently running at, a vector-based scheduling policy would have to be aware of the frequencies the tasks' activity vectors were sampled at, and would have to consider these frequencies when comparing activity vectors of tasks sampled at different frequencies.

The second option, defining activity vectors with respect to a fixed reference frequency, would mean that the activity vector of a task would not necessarily represent the actual resource utilization of the chip, but instead the hypothetical resource utilization the task would cause if the chip were operated at the reference frequency. For instance, a task T_1 could utilize 50% of the memory bandwidth when executed at the reference frequency, but only 30% when executed at a lower frequency. Thus, if the chip is currently running at the low frequency, scheduling T_1 will cause a memory bandwidth utilization of 30%, although the corresponding component of the task's activity vector has a value of 50%.

However, since our scheduling policies do not rely on the absolute values of the vector components, but rather compare activity vectors of different tasks, the drawback that activity vectors do not correspond to actual resource utilization at frequencies other than the reference frequency is no concern for us. Hence, we choose the second option and define activity vectors with respect to a fixed reference frequency. Abstracting activity vectors from the actual chip frequency facilitates the design of vector-based scheduling policies, since it results in activity vectors that are comparable regardless of the frequencies the corresponding tasks were running at when the activity vectors were sampled. As reference frequency for determining activity vectors, we choose the chip's maximum frequency.

For being able to determine activity vectors of tasks running at frequencies different from the maximum frequency, we supply a *translation vector* for each chip frequency. Similar to activity vectors, each component of the translation vector corresponds to one chip-related resource. We define the assignment of vector components to re-

sources to be the same for translation vectors as for activity vectors. A component of the translation vector denotes by which factor the utilization of a corresponding resource is expected to change when the chip is running at the respective frequency, compared to the maximum frequency. Thus, the components of the translation vector correspond to the slopes of the trend lines in Figure 5.8. We determine the translation vectors by running a set of representative benchmarks at all available frequencies and by calculating the average ratios of unit utilizations compared to the maximum frequency.

Using translation vectors, it is possible to "translate" resource utilization sampled at a specific frequency to an activity vector representing resource utilization with respect to the maximum frequency: For determining activity vectors, we sample the resource utilization of the chip at the actual chip frequency into a "raw" activity vector first, and do a component-wise division of the raw activity vector with the translation vector corresponding to the chip frequency in order to obtain the final activity vector.

5.5 Resource-conscious Scheduling

In Section 5.3, we have shown that avoiding resource contention is of paramount importance for achieving energy efficiency. We want to adapt timeslice-based multitasking, multiprocessor scheduling policies, like those found in today's general purpose operating systems, to become aware of and to avoid resource contention. While in the preceding sections, we have analyzed the impacts of resource contention using a CMP chip as an example, our policies proposed in the following sections are generic and suitable to avoid contention for other architectures with shared resources, such as SMT. For simplicity and without loss of generality, we will only talk of CMP in the following.

Since the degree of contention depends on the combination of tasks running simultaneously on the cores of a chip, we need to control the combination of tasks that run at a time. This leads to the concept of gang scheduling [Ous82]. While gang scheduling has been proposed to co-schedule threads of a multithreaded application in order to optimize communication, our goal is to combine tasks that use as different resources as possible. As already mentioned in Chapter 3, in this thesis, we only consider independent single-threaded tasks, i.e., we assume that there is no communication between tasks. If there is communication, co-scheduling based on communication patterns and co-scheduling based on resource utilization can be conflictive goals. This is a topic for future work.

A prerequisite for being able to co-schedule tasks is a suitable distribution of tasks to cores. In particular, tasks intended to be co-scheduled may not be assigned to the runqueue of one and the same core. In the following, we will introduce a migration policy that makes sure that tasks showing diverse resource utilizations are available on each processor for co-scheduling. Based thereon, we present two co-scheduling

policies, a specialized one that is applicable if there is one known resource mainly responsible for contention, and a generic one that is applicable without knowledge which resources are responsible for contention.

5.5.1 Vector balancing

We propose vector balancing as a task migration policy guided by activity vector information. The goal is to distribute tasks to cores in a way that enables co-scheduling tasks with different characteristics. Vector balancing uses a uniform view on activity vectors and considers all resources equally.

Vector balancing is comparable to the migration policies of activity balancing and activity unbalancing proposed in Chapter 4, in the sense that we calculate a metric from the activity vectors of all task in a runqueue, and strive to improve the metric by task migrations.

A simple solution for enabling co-scheduling would be to collect tasks with similar characteristics on one hardware context, for example, to run all memory-bound tasks on core 0 and all compute-bound tasks on core 1 of a dual-core processor (a policy similar to activity unbalancing). This way, even without a special co-scheduling policy, there would never be two memory-bound task running simultaneously on the chip.

However, it is not always optimal to collect similar tasks on one core. For example, several cache-intensive tasks on one core can overwrite each other's working sets in the cache, and memory-intensive tasks are always cache-intensive to a certain degree.

In addition, it is not always possible to divide tasks into sets that use mutually different resources, especially if there are only few resources to consider, as with multicore chips. For instance, assume a situation with two tasks of medium memory intensity, a memory-bound task and a compute-bound task (Figure 5.9). Collecting similar tasks would mean running the two tasks with medium memory intensity on one core. This would likely cause resource contention, since then the memory-bound task on the other core would have to be co-scheduled with one of the tasks with medium memory intensity, leading to a request for memory bandwidth greater than 100%.

The optimal solution would be to co-schedule both tasks of medium memory intensity, and to co-schedule the memory-bound and the compute-bound task. Although two tasks with medium memory utilization are still likely to interfere (see next section), the interference between two tasks of medium memory intensity is lower than the interference between a completely memory bound task and a task of medium memory intensity.

The same example also explains why a policy like activity balancing as proposed in Chapter 4 is no optimal base for resource-conscious co-scheduling. Activity balancing strives to level the average utilization of resources over the processors (cf. Section 4.4.2). In our previous example, on average, the memory utilization of the

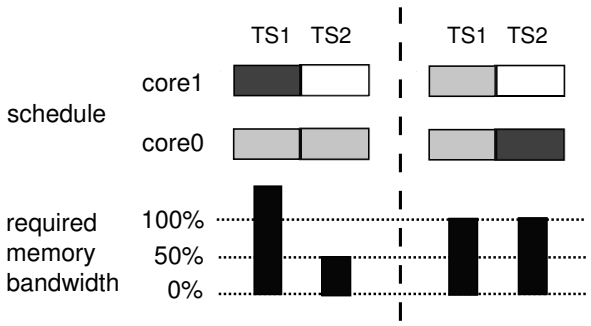


Figure 5.9: Sub-optimal (left) and optimal (right) co-scheduling of tasks. Shades of gray represent memory intensity.

tasks assigned to the cores is balanced. Yet, this does not prevent contention because of a lack of options what task to pair with the memory-bound task.

Instead of balancing utilization, we want to spread tasks with similar activity vectors over different cores (hence the term *vector balancing*), and collect tasks having different characteristics on each core. This gives co-scheduling policies more options for finding suitable combinations. In addition, for systems encompassing multiple multicore chips, applying the policy between cores belonging to different chips reduces resource contention by distributing tasks utilizing a certain resource to physically different chips.

Like with activity balancing in Chapter 4, we define a scalar metric that allows us to decide whether the migration of a task with a particular activity vector is beneficial or not. Our goal is to have tasks in a core's runqueue that show different degrees of utilization for the different chip units. In this respect, the statistical variance of the unit utilization among tasks in a runqueue constitutes a meaningful metric, since it allows to differentiate whether the tasks in a runqueue all show similar utilization of a particular chip unit, or if the utilization varies from task to task. Since we want a high variance in the utilization of all chip units, we use the sum over the variances of all vector components as a metric, which we define formally in the following way:

Definition. *Let x_i be a random variable describing the i–th component of the activity vector of a task selected at random from a particular core's runqueue, and $V(x_i)$ the statistical variance of the random variable x_i. Then our measure for balancing is the sum of all components' variances,* varsum*:*

$$\text{varsum} \;:=\; \sum_{i=1}^{n} V(x_i) \tag{5.1}$$

Example:

As an example, we assume a dual core. The runqueues contain tasks with the following activity vectors:

runqueue 0:
$$\begin{pmatrix} 0.1 \\ 0.2 \\ 0.8 \end{pmatrix}, \begin{pmatrix} 0.9 \\ 0.1 \\ 0.3 \end{pmatrix}, \begin{pmatrix} 0.8 \\ 0.2 \\ 0.2 \end{pmatrix}$$

runqueue 1:
$$\begin{pmatrix} 0.2 \\ 0.9 \\ 1.0 \end{pmatrix}, \begin{pmatrix} 0.1 \\ 0.7 \\ 0.8 \end{pmatrix}$$

For runqueue 0, the random variable x_0 can take the values of the first vector components (0.1, 0.9, and 0.8) with equal probability. This leads to a variance of $V(x_0) = 0.127$. The same way, we determine $V(x_1) = 0.002$, and $V(x_2) = 0.069$; thus $varsum = 0.198$. For runqueue 1, we obtain $varsum = 0.023$, accordingly.

Runqueue 1 consists of two similar tasks, hence it has a rather low *varsum* value. In addition, the last two tasks of runqueue 0 have similar characteristics, thus we could consider migrating one of them to runqueue 1 in order to improve the *varsum* of this queue (and thus give the scheduler of the corresponding core a greater variety of task characteristics to choose from).

After the migration, the runqueues could look the following way:

runqueue 0:
$$\begin{pmatrix} 0.1 \\ 0.2 \\ 0.8 \end{pmatrix}, \begin{pmatrix} 0.9 \\ 0.1 \\ 0.3 \end{pmatrix}$$

runqueue 1:
$$\begin{pmatrix} 0.2 \\ 0.9 \\ 1.0 \end{pmatrix}, \begin{pmatrix} 0.1 \\ 0.7 \\ 0.8 \end{pmatrix}, \begin{pmatrix} 0.8 \\ 0.2 \\ 0.2 \end{pmatrix}$$

The according *varsum* metrics are 0.225 for runqueue 0 and 0.298 for runqueue 1, which is an improvement for both queues. Thus, we perform the migration. The next step could be to migrate the second task of runqueue 1 to runqueue 0, for a final distribution of:

runqueue 0:
$$\begin{pmatrix} 0.1 \\ 0.2 \\ 0.8 \end{pmatrix}, \begin{pmatrix} 0.9 \\ 0.1 \\ 0.3 \end{pmatrix}, \begin{pmatrix} 0.1 \\ 0.7 \\ 0.8 \end{pmatrix}$$

runqueue 1:
$$\begin{pmatrix} 0.2 \\ 0.9 \\ 1.0 \end{pmatrix}, \begin{pmatrix} 0.8 \\ 0.2 \\ 0.2 \end{pmatrix}$$

Again, this improves the *varsum* of both queues (0.267 for runqueue 0 and 0.373 for runqueue 1), since it creates a greater variety in the middle vector components of runqueue 0, and removes one of two similar tasks from runqueue 1.

Migration policy

While in our example, the migrations we performed have always increased the *varsum* metric of both runqueues, there is also the case that a migration increases the *varsum* of one queue, but decreases the *varsum* of the other. Since our goal is having a reasonable amount of variance in all queues rather than maximizing the variance of a single queue,

we perform a migration if the minimum of the *varsum*s of both queues involved is greater after the migration than before. Like with activity balancing and unbalancing described in Chapter 4, we strive not to create load imbalances and migrate tasks back in exchange, if necessary. We also consider the *varsum* metric for the initial placement of newly started tasks.

Calculating the varsum metric

For optimizing the calculation of the *varsum* metric, we express the variance $V(x_i)$ in Equation 5.1 by using the expected values of x_i and x_i^2:

$$\text{varsum} = \sum_{i=1}^{n} E\left(x_i^2\right) - \left(E\left(x_i\right)\right)^2 \tag{5.2}$$

Since x_i is discrete (there is only a limited number of tasks in each runqueue), we can express $E(x_i)$ as $E(x_i) = \frac{1}{m}\sum_{j=1}^{m} a_{j,i}$, where m is the number of tasks in the queue, and $a_{j,i}$ is the i–th component of the j–th tasks's activity vector. The same applies to the squared values.

Thus, we can calculate *varsum* if we keep track of the sum of the activity vectors of tasks in a runqueue and of the sum of the vectors, squared per component:

$$\text{varsum} = \sum_{i=1}^{n}\left(\frac{1}{m}\sum_{j=1}^{m} a_{j,i}^2 - \left(\frac{1}{m}\sum_{j=1}^{m} a_{j,i}\right)^2\right) \tag{5.3}$$

5.5.2 Vector-based co-scheduling

As stated before, the goal of our co-scheduling policies is to run tasks that use mutually different resources together. Our policies build upon vector balancing as a policy that distributes tasks to cores according to their characteristics and accomplishes a heterogeneous collection of tasks on each core.

Although activity vectors provide degrees of resource utilization, this utilization cannot simply be used to 'divide' resources between tasks. For example, two tasks that both require 50% of the available memory bandwidth are likely to interfere although, in theory, their combined bandwidth requirements could be satisfied, since memory bursts of the two tasks can overlap [LVE00].

A similar problem exists for cache utilization. Firstly, we use access frequency as a proxy for space occupancy, which only correlates to a certain degree. Secondly, even if we used space occupancy as a metric, we could not preclude interference, since unless the cache is fully associative, conflict misses cause interference between tasks whose combined working sets would fit into the available cache space.

Hence, our proposed policies do not offer performance guarantees, but are best effort policies based on the rationale that tasks utilizing the same resources to a high degree slow down each other, and that we can reduce this slowdown by co-scheduling tasks that use different resources.

As motivated in Chapter 2, schedulers of today's general purpose operating systems make CPU-local, independent decisions. Rather than replacing these CPU-local schedulers by a global scheduling policy, our approach is to coordinate the decisions of the local scheduling policies. This way, we avoid the performance problems that global policies introduce, for example synchronization and locking overhead for shared data structures.

Coordinating CPU-local schedulers requires synchronizing the time of task switches and the actual choice of tasks to be scheduled. We use shared variables as a low-overhead mechanism for synchronizing task switches. For coordinating decisions, we make use of the principle of *runqueue sorting* introduced in Chapter 4.

We propose two approaches: *Sorted co-scheduling* is a non-uniform policy that is applicable if there is one resource known to be the main bottleneck. Sorted co-scheduling defines an order among tasks in each core's runqueue according to the utilization of the resource assumed to be the the bottleneck. *Greedy co-scheduling* is a uniform policy applicable if multiple resources cause contention, or if the main bottleneck is unknown. With greedy co-scheduling, each core can utilize resources that have been left over by previous scheduling decisions.

5.5.3 Sorted co-scheduling

For the case that there is one resource mainly responsible for contention, it is advantageous to concentrate on the one resource that causes contention instead of considering all resources. However, this requires knowledge about what resource is most important for avoiding contention, for example, that memory contention is more relevant than cache contention for typical application scenarios on the Intel Core2 processor. In terms of activity vectors, this means that we need a non-uniform view, being able to tell what component of the vector represents the most critical resource.

In the following, we present a co-scheduling policy that concentrates on a single resource. In case of the Core2 we use as an example chip, the most critical resource is memory bandwidth, but the policy in general is also applicable if any other single resource is responsible for contention. Without loss of generality, we use memory bandwidth as an example for the most critical resource.

The idea behind sorted co-scheduling is to use the utilization of a single resource, in our case memory bandwidth, to sort the tasks in each core's runqueue. Then we pair two cores, respectively, and select tasks with complementary demands for memory bandwidth for execution on the cores (Figure 5.10).

The policy assumes that the system is symmetric in the sense that on each level of the processor topology, the number of processors is a multiple of two. This as-

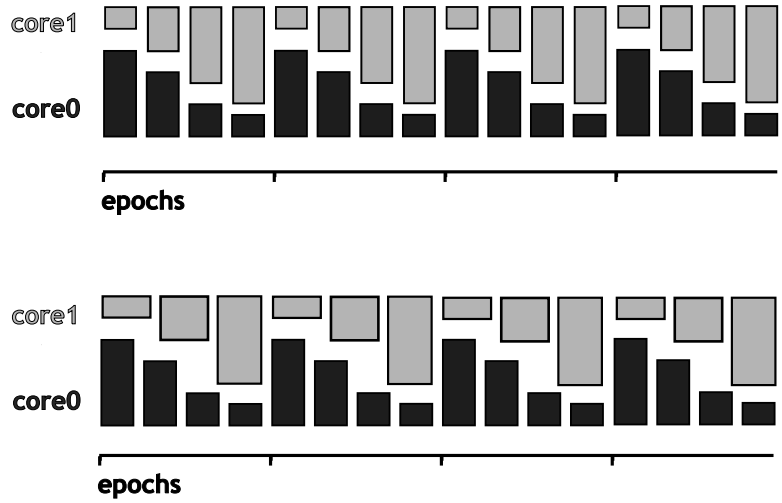


Figure 5.10: Sorted co-scheduling with equal (top) and unequal (bottom) runqueue lengths. Height of bars correspond to memory intensity.

sumption holds true in most of today's architectures. For instance, multithreaded Intel processors offer two logical threads per core (Hyper-Threading), and the Core2 quad-core we use as our test platform consist of two pairs of cores sharing a common L2 cache. Yet, there are exceptions, for example AMD's triple core Phenom or the triple core IBM PowerPC processor included in Microsoft's Xbox 360 [AB06].

To accomplish our goal of combining tasks with complementary demands for memory bandwidth, we define an order on the cores of a chip and assign consecutive numbers to the cores. We sort the tasks descendingly in runqueues of cores with even numbers and ascendingly in runqueues of cores with odd numbers. For sorting, we use the same lazy method described in Chapter 4. We divide the runqueue into an active and an expired part. For sorting descendingly, we always select the task among the first c tasks of the active queue that shows the highest value in the component of the activity vector that represents memory bandwidth utilization, and execute it (c is a constant). After execution, we queue the task into the expired queue. Eventually, this leads to a sorted queue, which we switch with the active queue whenever the latter is empty. Sorting ascendingly works accordingly by selecting the task with the lowest memory utilization among the first c tasks.

Co-scheduling requires synchronizing scheduling decisions across cores. To accomplish this, we divide time into epochs and synchronize the starting points of the epochs across all cores of a chip. We keep track of epochs by using a shared counter variable. The counter variable is incremented by *core 0*, the minimum of the order we defined on the cores, at the end of each epoch. The other cores read the shared vari-

able on each timer interrupt and, upon detecting a change, realize that a new epoch has started.

For sorted co-scheduling, we choose the epoch length to be m timeslices, where m is the number of tasks in the runqueue of the core that contains the most tasks. At the start of an epoch, on each core, we start to process the runqueue. Since the tasks in the runqueues are sorted in different directions, for cores whose runqueues contain the same number of tasks, this scheme leads to a constellation where tasks with complementary memory bandwidth demands are co-scheduled (top of Figure 5.10).

To account for runqueues containing different numbers of tasks, a situation that can occur despite load balancing if the total number of tasks is not divisible by the number of cores without remainder, we propose to increase timeslice lengths for runqueues containing fewer than m tasks. This does not unduly favor tasks in such queues, since those tasks already have a greater share of processor time than tasks in queues with m tasks regardless of the timeslice lengths.

During an epoch, the core with the most tasks can process its runqueue exactly once. To achieve the same property for queues with fewer tasks, we execute each task in a runqueue with $n < m$ tasks not for one standard timeslice, but for one nth of an epoch (which corresponds to for $\frac{m}{n}$ standard timeslices), as depicted in the bottom half of Figure 5.10.

Sorted scheduling requires little synchronization overhead, since synchronization need only happen after a runqueue is completely processed, and not on each task switch. If there are more than two cores on a chip, we shift the beginning of the epochs for each additional pair of cores. This way, situations when the first two cores both execute tasks with relatively low memory demands in the middle of their epoch can be used to run a memory-bound task on one of the remaining cores (see Figure 5.11).

This manner of interleaving is only possible if there is a sufficient number of tasks per core. If the number of tasks per core is smaller than the total number of cores, overlap cannot be avoided. For example, if there are four cores and two tasks per core, the most memory intensive task on core 0 will, because of sorting and coordination of epochs, always run together with the most memory intensive task of core 2.

5.5.4 Greedy co-scheduling

We propose greedy co-scheduling as a vector-based scheduling policy for avoiding resource contention. With greedy co-scheduling, cores of a CMP chip (or logical processors of an SMT chip) get to select tasks one after another, making use of the resources the other cores have left to them. In contrast to sorted co-scheduling, which focuses on one resource, greedy co-scheduling has a uniform view on activity vectors and considers all resources equally.

The aim of greedy co-scheduling is to even out the utilization of resources by combining tasks accordingly. The basic principle of greedy co-scheduling is that at each point in time, for each resource, the average utilization over the tasks co-scheduled on

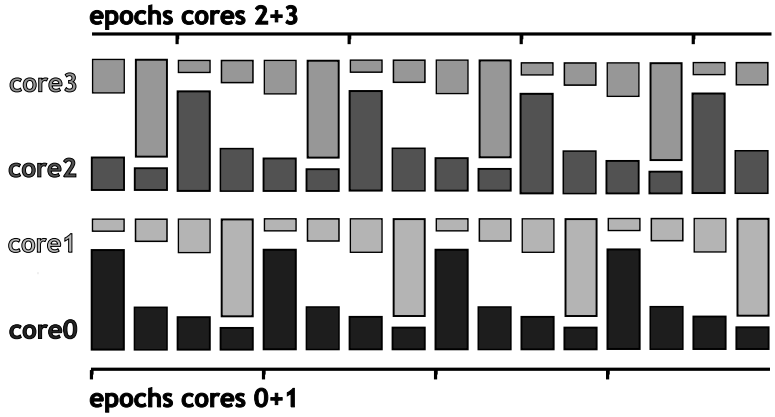


Figure 5.11: Sorted co-scheduling for more than two cores. Height of bars correspond to memory intensity.

the cores of a chip should be equal to the average utilization over all (ready-to-run *or* running) tasks assigned to the chip. The rationale behind this is that the entirety of tasks assigned to a chip defines the resource demands to that chip. These demands can be satisfied with the least contention if, during each timeslice, the same fraction of the demands is satisfied, or in other words, if the resource utilization is evened out over time.

As an example, if four memory-bound and four compute-bound tasks are assigned to a quad-core chip, instead of running four memory-bound tasks during one timeslice and then four compute-bound tasks during the next, the demand for memory bandwidth should be evened out over the timeslices and two memory-bound and two compute-bound tasks be combined each timeslice (see Figure 5.12, in the middle). Thus, if half of the total workload assigned to a chip is memory-bound, contention can be reduced if, during each timeslice, also half of the tasks co-scheduled are memory-bound. If, on the other hand, the workload consisted of two memory-bound and six compute-bound tasks, each timeslice, only one quarter of the tasks co-scheduled (i.e., one task) should be memory-bound (Figure 5.12, on the left). The same holds true for all other resources.

Greedy co-scheduling is based on the average resource utilization of the tasks assigned to a chip. Hence, in order to perform greedy co-scheduling, we introduce the *average vector* $\vec{A}$, which we define as the component-wise average of all activity vectors of tasks assigned to a chip:

$$\vec{A} = \frac{1}{m} \sum_{i=1}^{m} \vec{a}_i \tag{5.4}$$

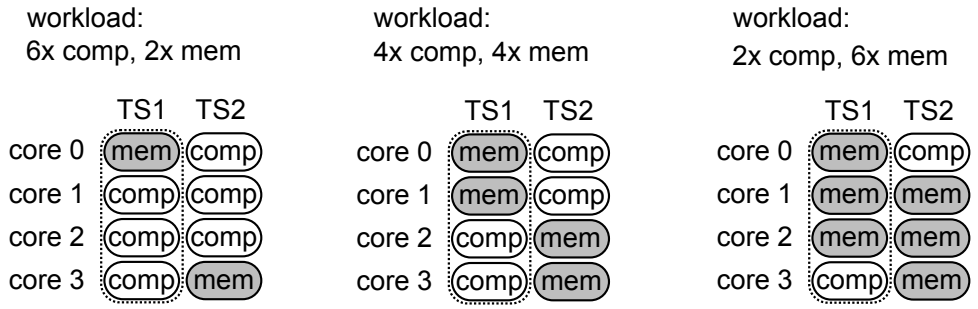


Figure 5.12: Optimal co-scheduling: The optimal number of tasks with a certain characteristic co-scheduled during a timeslice depends on the entire workload.

where m is the number of tasks on the chip and $\vec{a}_i$ with $i \in \{1, \ldots m\}$ are the activity vectors of the tasks assigned to the chip. Since in order to perform vector balancing, we already keep track of the component-wise sums of activity vectors for each runqueue (see Section 5.5.1), we can easily compute the average for the entire chip by doing a component-wise summation of the already aggregated per-runqueue sums, followed by a division by the total number of tasks in the chip's runqueues.

As explained above, in order to even out resource utilization over the timeslices, our goal is that during each timeslice, the average of the activity vectors of the tasks co-scheduled should be as close as possible to the average vector just defined. Like for sorted co-scheduling, we use the concept of epochs to synchronize scheduling decisions. For greedy co-scheduling, we choose the epoch length to be equal to the length of one standard timeslice, and initiate a task switch on all cores at the start of each epoch. This way, all task switches are synchronized. If tasks do not block or terminate, all cores switch tasks at nearly identical points in time, limited by only the skew of timer interrupts between the cores.

To coordinate the task selections of the individual cores of a chip, the cores get to choose which task to execute one after another according to an order we define on the cores analogously to sorted co-scheduling. We use a shared variable denoting which core is allowed to make its selection.

To keep track of the resources utilized by the tasks that are selected for execution on the individual cores, we introduce the *allocation vector* $\vec{s}$. The allocation vector is the sum of the activity vectors of the tasks selected for execution so far. According to our aim of evening out resource utilization, on each core, we select the task to be executed in a way that the allocation vector, divided by the number of cores that have already made their selection, comes as close to the average vector as possible (Figure 5.13).

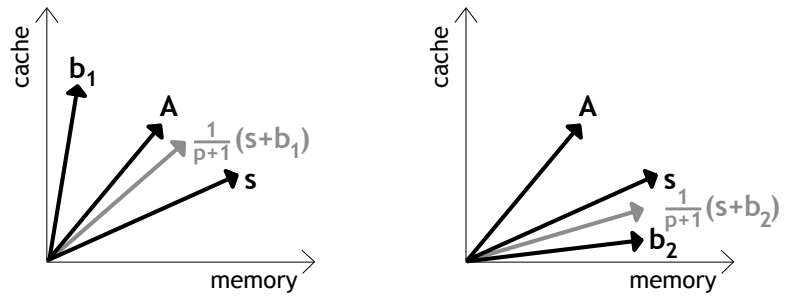


Figure 5.13: Selection process of greedy co-scheduling. Activity vector $\vec{b}_1$ (left graph) is more suitable than $\vec{b}_2$ (right side), since it results in a smaller difference to the average vector $\vec{A}$.

The process of task selection starts with core 0. Since core 0 is the first core to make a scheduling decision, we consider no resources as occupied yet and need not select a task having specific characteristics on core 0, but can select any task. Thus, at the beginning of an epoch, core 0 just selects the task at the head of its runqueue for execution and initializes the allocation vector $\vec{s}$ with the activity vector $\vec{a}$ of the selected task:

$$\vec{s} := \vec{a}$$

Then core 0 increments the shared variable, denoting that now the next core, *core 1* can make its selection. On core 1 and all following cores, the selection process is the same, thus, in the following, we will speak of *core p* as the core that is currently selecting a task.

Again, to limit the overhead for task selection, we only consider the first c tasks in a runqueue for greedy co-scheduling. Hence, core p looks at the first c tasks in its runqueue and chooses the one that results in a resource utilization on the chip that comes as close as possible to the average. Thus, we check how close the average vector we would get if we added the activity vector $\vec{b}_i$ of the task under consideration to the allocation vector $\vec{s}$. We use the Manhattan distance (see Section 4.4.1) between the average vector $\vec{A}$ and the average of the tasks currently selected plus the task considered for selection. Since at the moment, p cores have already made their selection, and their activity vectors are contained in $\vec{s}$, we can calculate this average as $\frac{1}{p+1}\left(\vec{s}+\vec{b}_i\right)$.

Thus, if $\vec{b}_i$ with $i \in \{1,\ldots c\}$ are the activity vectors of the tasks we consider, n is the dimension of the activity vectors, and p is the number of the core making the

selection, we choose the task with activity vector $\vec{b}_j$ with

$$j = \arg\min_{i=1}^{c} \left|\left| \vec{A} - \frac{1}{p+1}\left(\vec{s}+\vec{b}_i\right) \right|\right|_{\text{manh}} \tag{5.5}$$

Then we add the selected task's activity vector to the resource vector

$$\vec{s} := \vec{s} + \vec{b}_j$$

and increment the shared variable, denoting that core $p+1$ can make its selection. This continues for all cores on the chip.

To avoid starvation, all tasks in a runqueue need to be selected at least once before the same task is eligible again. Like with runqueue sorting and sorted co-scheduling, we divide the runqueue into an active and an expired part for this purpose. Also similar to runqueue sorting, our method eventually arranges the runqueues in a fashion that there is a high probability that a suitable task can be found among the first c tasks in each queue.

5.5.5 Scalability

As mentioned, sorted co-scheduling requires little synchronization, since not the individual task switches, but only the epochs need be synchronized, which for sorted co-scheduling last for a number of timeslices corresponding to the number of tasks in the longest runqueue.

For greedy co-scheduling, an epoch lasts only one timeslice, and hence, the individual task switches need be synchronized. After a new epoch has started, a core must wait on all its predecessors before it can make the next scheduling decision. Since timer interrupts need not necessarily arrive at the same time on different cores, in the worst case, a timer tick passes between the scheduling decisions of two successive cores. This is tolerable if only a few cores share resources, and timer ticks are short in comparison to timeslice lengths (for example, 1ms timer ticks and 100ms timeslice lengths). In addition, cores are not idle while they wait on the other cores to make their scheduling decisions, but continue executing the task from the preceding epoch. The only consequence is that for at most x timer ticks, where x is the number of cores synchronizing scheduling decisions, there can be a sub-optimal combination of tasks running.

For both policies, synchronization of scheduling decisions needs only be applied to processors that actually share resources, e.g., cores or logical processors of the same chip, or, in case of memory bandwidth, processors of the same NUMA node. Typically, by hardware design, only a limited number of processors are sharing resources, lest the shared resources become a severe bottleneck. This allows us to accomplish scalability by providing the shared variables required for our algorithms (epoch

counter, resource vector) separately for each set of processors sharing resources, e.g, for each NUMA node or each chip.

5.6 Frequency Heuristic

In Section 5.3, we have pointed out that the primary lever to achieve energy efficiency is to combine tasks in a way that avoids resource contention, and that co-scheduling tasks that profit from frequency scaling is not beneficial. However, for inauspicious workloads that contain too many memory-bound tasks, our scheduling policy must co-schedule memory-bound tasks owing to a lack of compute-bound tasks to combine them with. Thus, situations in which it is beneficial to engage frequency scaling can still occur. Even though we do not co-schedule memory-bound tasks if this can be avoided, we use frequency scaling as a fallback solution for cases in which we cannot prevent contention because of an inauspicious workload that contains too many memory-bound tasks.

Sorted or greedy co-scheduling, although designed primarily to avoid contention, also facilitate frequency selection, since the scheduling policy controls the combination of tasks running at a time. Thus, scheduling decisions do not occur randomly and independently on the cores, and we can choose a frequency fitting the characteristics of the currently running tasks.

Heuristics for frequency selection found in the literature that are based on task characteristics have used the metrics of memory intensity and on-chip activity [WB02, KDG+04, DR07] for deciding about the optimal frequency. Snowdon et al. [SPH07, SvdLPH07] has shown that the runtime of and the power consumed by a particular task at a certain frequency can be predicted using several architecture-specific performance counter metrics, which in turn can be used to infer the optimal frequency for running a task.

Since our focus is not on finding a sophisticated heuristic, but rather on the interactions of resource contention and frequency scaling, we use a rather simple heuristic based solely on the metric of memory intensity. We use the utilization of memory bandwidth caused by a task to predict how frequency scaling would influence the EDP. We introduce a measure we call *EDP factor*. The EDP factor of a task denotes the factor by which the EDP of the task changes when the task is executed at a certain frequency f, relative to the EDP at the highest frequency $f_{\max}$.

Our model to predict the EDP factor $factor_f(x)$ of a task with a certain memory bandwidth utilization x for a given frequency f is based on linear interpolation between two reference values. $factor_f(0)$ is the EDP factor of a completely compute-bound task ($x = 0$, no memory bandwidth utilization), and $factor_f(1)$ is the EDP factor of a completely memory-bound task ($x = 1$, maximum memory bandwidth utilization). To obtain $factor_f(x)$, we interpolate between these values (see Figure 5.14). Thus, we calculate the EDP factor of a task with memory bandwidth utilization x at

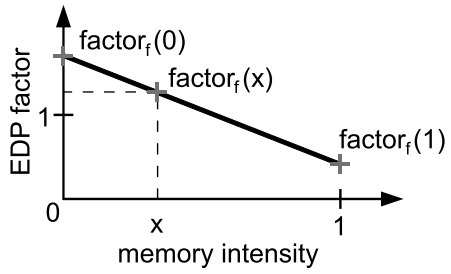


Figure 5.14: Interpolation of the EDP factor

the frequency f the following way:

$$\text{factor}_f(x) = x \cdot \text{factor}_f(1) + (1-x) \cdot \text{factor}_f(0) \tag{5.6}$$

We determine the reference values $\text{factor}_f(0)$ and $\text{factor}_f(1)$ by measuring EDP for two microbenchmarks that cause no memory utilization and maximum memory utilization at the frequencies f and $f_{\max}$.

Note that using linear interpolation assumes a very simple processor. It does not account for features that modern processors possess for hiding memory latencies, such as out-of-order execution with outstanding loads or write buffers. However, with a small adaption of the reference values (see Section 5.7.3), we found this simple model to be accurate enough for our purpose of evaluating the interactions of co-scheduling and a frequency policy.

To apply frequency selection based on the EDP factor, after each scheduling decision, we check which frequency would yield the lowest EDP factor, averaged over the tasks currently selected for execution on the cores of a chip, and set the frequency of the chip accordingly.

Although frequency transitions do not halt execution for a long time on modern processors, we strive to minimize the number of frequency transitions, since especially stabilizing the voltage to a new level after a transition can take a considerable amount of time (up to milliseconds with commonly used off-chip voltage regulators [KGWB08]). Typically, computation can be performed during the voltage switch, so this does not diminish performance, but does reduce energy savings.

With greedy co-scheduling, the individual cores select their tasks one after the other. To avoid unnecessary frequency changes, we only adapt the frequency after the last core has made its decision. Similarly, for sorted co-scheduling, after a task switch has taken place on a core, we check whether a timeslice is likely to expire on any core of the same chip soon, and only change the frequency if this is not the case.

5.7 Implementation

We implemented task activity vectors, our proposed scheduling strategies, and the frequency heuristic for the Intel Core2 architecture and the Linux 2.6.22 kernel.

5.7.1 Activity vectors

In the following, we describe briefly how we determine the utilization of the resources represented by activity vectors for the Intel Core2 architecture.

Memory bandwidth For determining memory bandwidth utilization, we count the number of bus transactions initiated by a core during a timer tick (period of time between two successive timer interrupts) using the event `BUS_TRANS_ANY`, and divide that number by the theoretical maximum number of transactions the hardware supports during this period of time.

The theoretical maximum is determined by the front side bus and the memory and hence depends not only on the processor, but also on the memory deployed in the system. Our test system has a front side bus supporting 1066MT/s (MT/s = millions of transactions per second) and DDR2 PC-6400 RAM supporting 800MT/s. As a consequence, the transfer rate is limited by the ram and is 800MT/s at most. In practice, we observed transfer rates of up to 720MT/s using the `stream` memory benchmark.

The bus can transfer eight bytes in parallel, but the system always transfers complete cache lines of 64 bytes and the corresponding performance monitoring counter considers this as one transaction. Therefore, the maximum event rate for bus transactions in our test system is $800\text{MT/s} \cdot \frac{8}{64} = 100\text{MT/s}$, amounting to one transaction every 24 processor cycles, assuming a 2.4GHz processor clock.

L2 Cache On the Core2 architecture, there is a large number of events available for counting requests to the L2 cache, including loads, stores, invalidations, and prefetch requests. Since there are not enough counters to cover all events separately, we use the event `L2_RQSTS`, which accumulates all types of requests. However, it is hard to give a theoretical maximum for this event, since different kinds of requests have different maximum rates (for example, a transfer from memory to L2 takes longer than a transfer from L2 to L1).

To get a practical upper limit for L2 activity, we ran the `stream` memory benchmark, but with the memory size reduced to fit completely into the L2 cache (`stream-fit1`, cf. Section 5.3.4). We measured the number of L2 references during the run, which amounted to one reference every four cycles, so we chose this value as maximum.

Non-shared Rather than counting events for the various chip units (which is difficult because of a limited number of performance counters in the Core2 architecture), we use the number of retired instructions (event `INSTR_RETIRED_ANY`) as a proxy for the activity of the non-shared resources. This neglects various aspects, e.g., that different instructions keep the chip busy for different amounts of time, and that some instructions do not retire because of misprediction, but cause chip activity nonetheless.

To obtain utilization, we divide the number of retired instructions by the total number of processor cycles. Since the chip can execute micro-operations in parallel, the number of retired instructions per cycle can be greater than one, meaning that the count of retired instructions is greater than the cycle count. We define chip activity to be at 100% if the number of retired instructions is equal to or greater than the cycle count.

Overall, this is only a very rough estimation for the utilization of the resources of a core that are not shared with other cores. Yet, it is sufficient for our purposes: Typically, memory or cache-bound tasks, which do not utilize other resources heavily, show a low number of retired instructions per cycle, while tasks that show low memory and cache utilization and whose speed is bounded by resources not shared between cores show a higher number of retired instructions per cycle.

Frequency dependency

Each component of an activity vector is defined as the utilization of a particular resource, i.e., as the ratio between the number of requests issued to the resource during a period of time divided by the maximum number of requests the resource supports during that period of time.

As mentioned in Section 5.4.1, depending on the resource, the maximum number of requests a resource supports per time can or cannot be dependent on the frequency the chip is running at. For example, the maximum number of requests memory can service in a period of time does not depend on the chip frequency, assuming that the bus frequency is independent from the chip frequency. In contrast, the maximum number of requests an on-chip cache or an ALU can service per time decreases when the chip frequency is scaled down. We need to take this into account when calculating resource utilization, since we always determine resource utilization with respect to the maximum number of accesses a resource supports during a period of time.

Most processors encompass a timestamp counter (TSC), a special register that counts processor cycles. On the Core2 architecture, the TSC does not get scaled with the chip frequency, i.e, when frequency scaling is applied, the TSC keeps counting cycles at the reference frequency (in our case, 2.4GHz). This makes the TSC usable for calculating the utilization of resources whose maximum access rate is independent from the chip frequency.

For counting scaled cycles, we use a performance monitoring counter. We use the event `CPU_CLK_UNHALTED_CORE`, which counts unhalted core cycles and gets scaled

with the chip frequency. In contrast to the time stamp counter, this event only increments when the chip is not idle. However, this is not a problem. An idle core is typically put into a low-power sleep mode, and no task, or on some operating systems such as Linux, a specific idle-task is running (depending on whether the operating system executes the instructions to put an idle chip into low-power mode within task context). In any case, in the low-power mode, no resources are utilized, so we do not need to obtain any event counts.

Having counters for scaled and unscaled cycles allows us to correctly calculate the utilization of resources whose maximum number of requests per time is dependent on or independent of the chip frequency. For calculating memory bandwidth utilization, we use the unscaled cycle count as reference, whereas for calculating the utilization of L2 cache and the non-shared resources, we use the scaled cycle count.

In addition, as described in Section 5.4.1, we apply a correction vector to activity vectors sampled at 1.6GHz to predict the behavior of the applications at the full frequency of 2.4GHz.

5.7.2 Vector balancing and co-scheduling

For implementing vector balancing, we modified Linux's load balancing algorithm to consider our measure of variance as defined in Section 5.5.1 next to load. The implementation is the same as for activity balancing described in Section 4.5.3, except that we use a different metric. Again, we use fixed-point arithmetic for all metrics involved in vector-based scheduling.

Introducing co-scheduling into the Linux scheduler requires only few modifications. We introduce an epoch counter as an atomic shared variable. With greedy co-scheduling, the epoch counter is incremented whenever a number of timer ticks corresponding to one standard timeslice has passed. We use a 1000Hz timer and 100ms timeslices; hence in our case, we increment the epoch counter every 100th timer tick. With sorted co-scheduling, the epoch counter is incremented whenever a number of timer ticks corresponding to m standard timeslices has passed, where m is the number of tasks in the longest runqueue. Thus, we increment the epoch counter every $m \cdot 100$th tick. For greedy co-scheduling, we additionally implement a shared variable for denoting which core is supposed to make its scheduling decision and implement the average vector and the allocation vector as shared memory buffers.

To apply co-scheduling, we need to replace the logic that checks whether a task's timeslice has expired in order to coordinate task switches across cores or siblings. For greedy co-scheduling, we initiate a task switch whenever an epoch change takes place. For sorted co-scheduling, we switch tasks whenever one nth of an epoch has passed, where n is the number of tasks in the respective runqueue.

A special case that needs to be considered are tasks that block or unblock. Tasks that terminate or are newly started fall into the same category. While our co-scheduling policies are not specifically tailored to interactive tasks, any implementation of a

scheduling policy must be able to handle the case of changes in the set of runnable tasks.

Whenever a task unblocks or is newly started, the number of tasks in a runqueue increases. With greedy co-scheduling, this is not a problem. When the next scheduling decision is made, there is simply one more task eligible. With sorted co-scheduling, we decrease the amount of time allotted to each task for the remainder of the epoch in order to be able to execute all tasks within the epoch. Similarly, we increase the time each task may run with sorted scheduling when a task leaves the runqueue (upon blocking or termination). With greedy scheduling, we simply choose the next task of the runqueue when the currently executed tasks blocks or terminates. We do not specifically consider activity vectors for choosing the next task in this case, since the focus of our scheduling policies is on CPU-bound tasks.

5.7.3 Frequency selection

Under Linux, the processor frequency is controlled by a governor framework which allows selecting a policy for frequency scaling from user space. The framework is not suitable for initiating frequency switches from the scheduler; in particular, the functions provided by the device drivers controlling the chip frequency are not intended to be called from the scheduler, but only from the governors running in task context.

We therefore deactivated the framework and the driver, and implemented our own prototype method for selecting a suitable frequency. To set the chip frequency according to the decisions of our frequency heuristic, we directly program the model specific registers of the Core 2. Ultimately, it would be beneficial to adapt the device drivers to support frequency changes initiated by the scheduler.

After a task switch, if no switch is likely to occur on any other core on the same chip, we check how the EDP of each task currently running on the chip would scale according to the EDP factor as defined in Section 5.6 and calculate the average of the EDP factors of all tasks. As explained in Section 5.3.5, we consider two frequencies, 2.4GHz and 1.6GHz. If the average EDP factor is lower for 1.6GHz than for 2.4GHz, we scale the frequency down to 1.6GHz, otherwise we use the maximum frequency of 2.4GHz.

As mentioned in Section 5.6, we use microbenchmarks to calibrate the two reference EDP factors. The experiments with the microbenchmarks presented in Section 5.3 indicate that for our test chip, on average, the EDP of tasks with memory utilization of 0% scales by a factor of 1.67 when lowering the frequency, whereas the EDP of tasks with memory utilization of 100% scales by factor of 0.88 (Table 5.1). Based on this, we would estimate the scaling of EDP of a task with memory utilization x according to Equation 5.6:

$$\text{factor}_{1.6}(x) = x * 0.88 + (1 - x) * 1.67 = 1.67 - 0.79x \tag{5.7}$$

In practice, we observed that using this estimation prevented frequency scaling in situations where it would have been beneficial.

We mentioned in Section 5.6 that modern processors are able to hide memory latencies to a certain degree and thus can have several memory requests outstanding and still continue executing instructions not dependent on the results of the memory references. As a result, the degree of slowdown that results from limited memory bandwidth depends on the instruction-level parallelism the task exhibits. Our benchmark used for calibration, the `stream` benchmark, performs loops over arrays, and the individual iterations are independent from each other, resulting in high instruction-level parallelism. Real-world applications can show less instruction-level parallelism, resulting in a lower EDP factor. (They benefit from lower frequencies already at lower memory utilizations than interpolated from the microbenchmark.)

Considering this effect despite our very simple model requires fine-tuning the model parameters. Engaging frequency scaling already for tasks with lower memory utilization can be accomplished either by reducing the constant or the proportional part of Equation 5.7. We performed experiments with several memory-intensive SPEC benchmarks and found modifying the constant part while only slightly changing the proportional part to be a viable solution. For our experiments presented in the next sections, we use the following estimation of the EDP factor:

$$\text{factor}_{1.6}(x) = 1.6 - 0.8x \tag{5.8}$$

5.8 Evaluation

We evaluated our implementation on the Intel Core2 Quad described in Section 5.3.1. For power measurements, we used the measurement infrastructure described in Section 5.3.2.

5.8.1 Methodology

On a multicore processor, the runtime of a benchmark depends strongly on the characteristics of the other tasks that are co-scheduled with it. With the standard Linux scheduler, the combination of tasks running at a time is arbitrary. Therefore, the performance of a benchmark and the amount of energy required to run it can vary considerably between different runs of the benchmark. In addition, different benchmarks take different amounts of time to complete. To create a constant workload and to be able to compensate statistical effects, we ran all benchmarks in loops (each benchmark at least ten times) and used the average runtime and energy consumption sampled over all runs for evaluation.

benchmark	standard	avector	sorted	greedy
gamess	1.000	1.001	0.996	1.001
gobmk	1.000	1.002	0.995	0.998
gromacs	1.000	0.997	1.006	1.006
hmmer	1.000	1.002	0.993	9.999
namd	1.000	1.000	1.007	1.001
povray	1.000	1.001	1.003	1.001
sjeng	1.000	0.999	1.000	1.002
tonto	1.000	1.000	0.998	0.998
average	**1.000**	**1.000**	**1.000**	**1.001**

Table 5.5: Overhead of vector-based scheduling—normalized runtime of compute-bound benchmarks

5.8.2 Overhead

We evaluated the overhead of our implementation of activity vectors for the Core2 architecture and the overhead of our vector-based co-scheduling policies by running a workload composed solely of compute-bound SPEC benchmarks. This workload constitutes a worst-case for our co-scheduling policies, since a compute-bound task's runtime does not depend on co-scheduling (cf. Section 5.3.4), and hence no improvements can be achieved by our policies. Thus, any overhead of providing activity vectors and of performing co-scheduling should become apparent.

Table 5.5 shows runtimes of a workload composed of eight compute-bound benchmarks that showed no interference via shared resources in our analysis in Section 5.3.4. We measured the runtime of the benchmarks using a standard Linux kernel, a kernel that provides activity vectors but does not use them for scheduling, and kernels that apply sorted and greedy co-scheduling, respectively. The co-scheduling policies also include vector balancing, but not the frequency heuristic, whose overhead we will evaluate separately in Section 5.8.6. All benchmark runtimes are normalized to the runtime of the respective benchmark under standard Linux.

Neither activity vectors nor the co-scheduling policies cause measurable overhead. The fact that some benchmarks show a slightly reduced runtime in some setups shows that the overhead lies within the measurement tolerance, i.e., is smaller than the effects of other factors that influence the benchmarks' runtime during the measurements. (For instance, a changed memory layout caused by changed environment variables or address space randomization can cause benchmark runtimes to vary [WM08,MDHS09].)

The reason why our implementation of activity vectors for the Core2 architecture, in contrast to the implementation for the NetBurst architecture (Chapter 4), does not

show significant overhead is that the vectors for the Core2 architecture possess only three components and involve only four performance monitoring counters and no multiplexing.

5.8.3 Workload dependence

Ratio of memory-bound to compute-bound tasks

The benefits that our scheduling policies can achieve depend strongly on the composition of the workload running on the system. To investigate the dependency of vector-based co-scheduling on the workload, in particular, on the ratio of compute-bound to memory-bound tasks, we composed different workloads of the microbenchmarks introduced in Section 5.3.4 (the compute-bound `aluadd` and the memory-bound `stream` microbenchmark). All workloads consist of eight tasks. We varied the number of memory-bound tasks in the workload between one and eight, while filling the remaining slots with compute-bound tasks. We ran all workloads using the standard Linux scheduler, using sorted co-scheduling, and using sorted co-scheduling in combination with the frequency heuristic. We also performed the experiments with greedy co-scheduling instead of sorted co-scheduling, but since the results are similar, we only present the results of sorted co-scheduling.

First of all, we noted in our experiment that the runtime of the compute-bound `aluadd` benchmark hardly varies for the different workloads and scheduling policies, which is consistent with our observations from the experiments in Section 5.3.4. In the following, we only discuss the more interesting results of the memory-bound `stream` benchmark.

Figure 5.15 shows the effect our co-scheduling policy has on the runtime of `stream`. The figure depicts the runtime of a single instance of `stream` for the scenarios under standard Linux scheduling and sorted scheduling. With standard Linux scheduling, there is a steady increase in runtime per `stream` instance when the number of instances is increased. Recall that the total number of tasks is constantly eight (the remaining slots are filled with `aluadd`), so the increase in runtime stems purely from the memory contention between the `stream` instances.

While sorted scheduling cannot avoid this increase of runtime in general, for certain workload scenarios, sorting reduces contention and runtime by co-scheduling instances of `stream` with instances of `aluadd`. The most significant reduction of runtime (reduction by 20%) is achieved for a workload containing two instances of stream, since for this scenario, sorted co-scheduling can completely avoid memory contention by co-scheduling each instance of `stream` with three of the six `aluadd` instances, while standard Linux schedules both `stream` instances at the same time in many cases.

With more than two instances of `stream`, runqueue sorting also has to co-schedule some instances of the memory-bound `stream` benchmark, so the improvements over

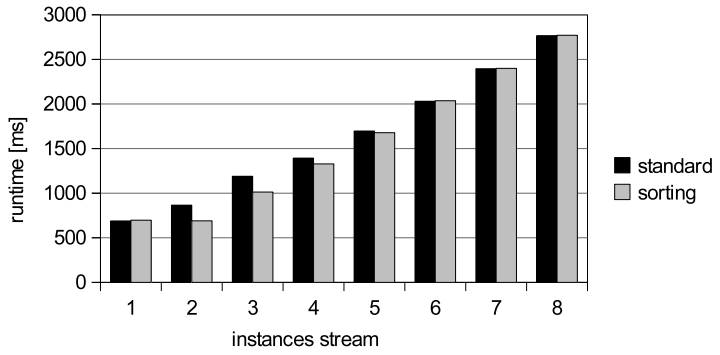


Figure 5.15: Runtime of one stream instance for different workloads

standard Linux scheduling diminish with the number of `stream` instances contained in the workload. However, a significant improvement persists for workloads containing up to four `stream` instances. The reason is that for such workloads, standard Linux scheduling has the potential of inadvertently wasting memory-bandwidth by running `aluadd` on all four cores during some timeslices, which is no longer possible for workloads that contain five or more instances of `stream` (and thus three or less instances of `aluadd`).

Figure 5.16 shows in detail how sorted co-scheduling influences the combination of memory-bound and compute-bound tasks co-scheduled. For each setup, the corresponding diagram displays how often (percentage of time) each combination of memory-bound and compute-bound tasks occurred. Since our experiment ran on a quad-core system, the possible combinations range from 0 memory-bound and 4 compute-bound tasks to 4 compute-bound and 0 memory-bound tasks co-scheduled at a time. With standard Linux scheduling, the occurrence of the respective combinations is distributed probabilistically, while sorted co-scheduling always selects the combination that leads to minimal contention.

This experiment also shows the margin of the performance improvements our policies can achieve. The analysis of Section 5.3.4 indicated that resource contention can degrade performance by up to a factor of four on our quad core. However, we cannot expect our policies to improve performance on a similar scale. There are two factors that determine the improvement in performance our policies can yield for a given scenario: (1) the amount of contention that occurs under a resource-oblivious scheduling policy, and (2) the amount of contention that necessarily occurs, even under an optimal resource-conscious policy. Our co-scheduling policies yield maximum improvements if factor (1) is high and factor (2) is low, i.e., if a resource-oblivious policy causes much contention while a resource-conscious policy yields little contention. Factor (1) is maximized for workloads containing many memory-bound tasks, while factor (2)

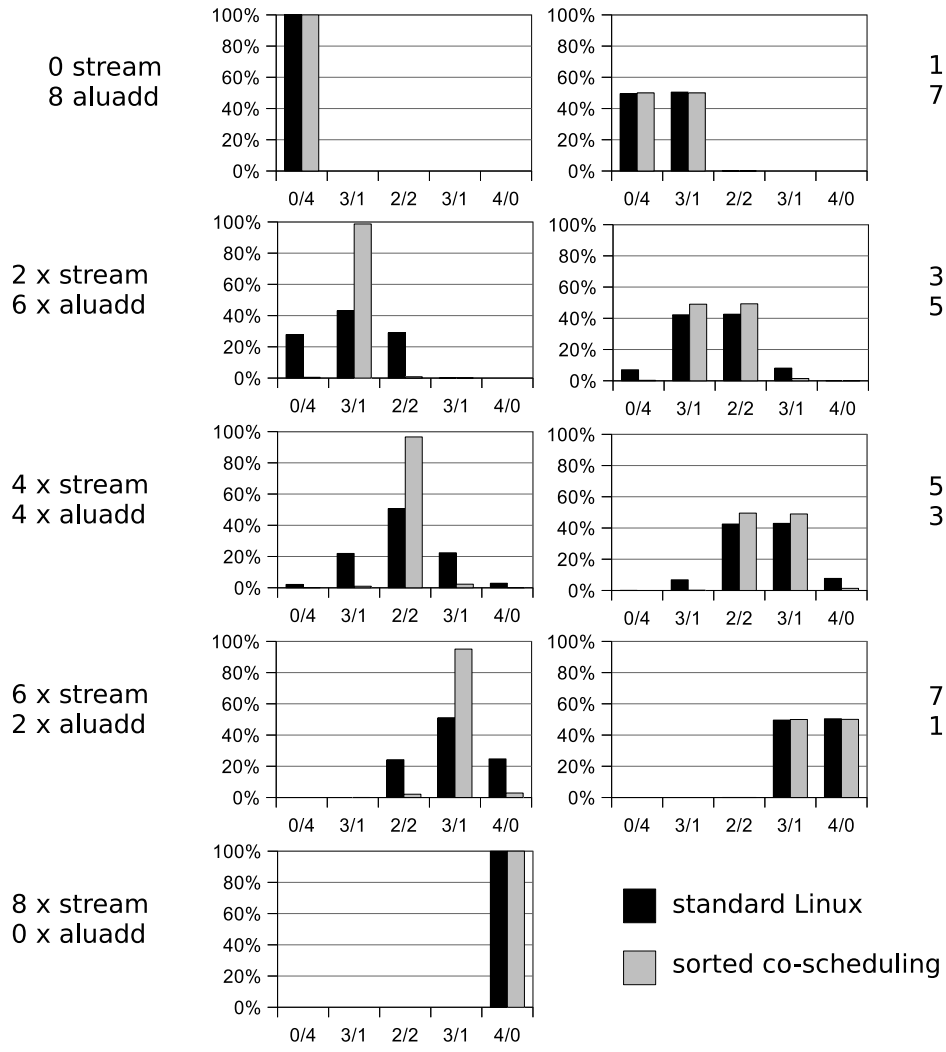


Figure 5.16: Task combinations with standard Linux scheduling and sorted co-scheduling.
x-axis: combination (#stream/#aluadd) of tasks co-scheduled
y-axis: occurrence

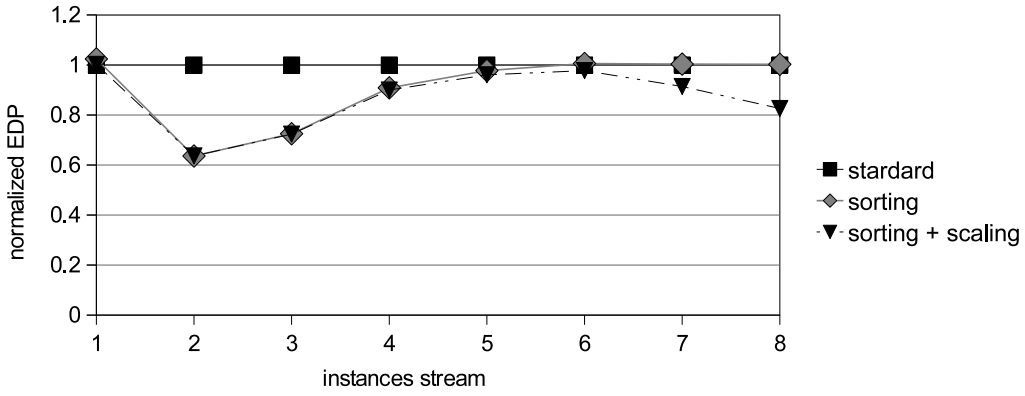


Figure 5.17: Normalized EDP of stream for different workloads

is minimized for workloads containing only few or no memory-bound tasks. Thus, no workload can at the same time maximize factor (1) and minimize factor (2). For the mixed workloads for which our policies can yield an improvement, neither does resource oblivious scheduling cause the maximum possible contention, nor is it possible to completely avoid contention in all cases, which limits the margin for improvements.

As mentioned, maximum benefits are achievable for the scenario with two memory-bound and six compute-bound tasks, since this is the only scenario for which a schedule is possible that completely avoids co-scheduling tasks that would cause contention. However, in this case, the degree of contention introduced by standard Linux scheduling is still relatively moderate. On the other hand, for the scenarios with more memory-bound tasks, which cause more contention under standard Linux scheduling, some memory-bound tasks have to be co-scheduled even under resource-conscious scheduling.

We also investigated the impact of our scheduling policies on EDP, which considers both runtime and energy consumption. Figure 5.17 displays the normalized EDP of a `stream` instance under standard Linux scheduling, sorted co-scheduling, and sorted co-scheduling in combination with the frequency heuristic. The figure confirms again that vector-based co-scheduling yields improvements for workloads containing two to four `stream` instances.

In addition, the experiment shows that the frequency heuristic is only beneficial if co-scheduling is not, and vice versa. For one to six instances of `stream`, the EDPs for sorted co-scheduling with and without the heuristic are nearly identical. Only for seven and eight instances of `stream`, the frequency heuristic yields improvements. The reason is that workloads with up to six `stream` instances include at least two instances of `aluadd`, so with eight tasks total and four cores, co-scheduling chooses an `aluadd` instance every timeslice on some core. The huge penalty `aluadd` would

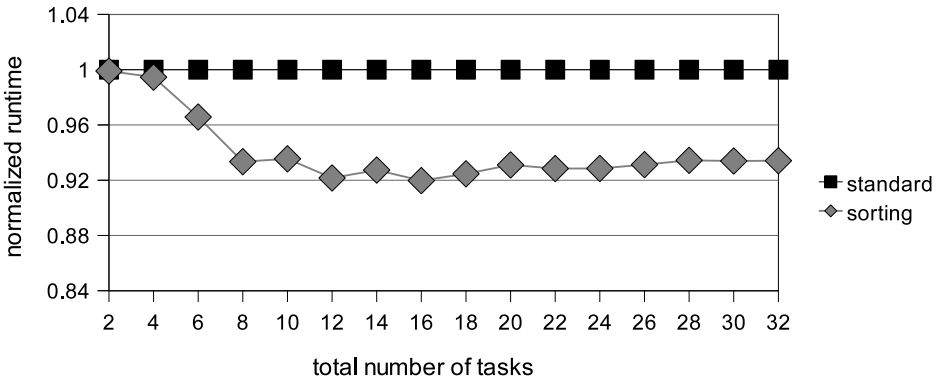


Figure 5.18: Normalized runtime of stream for different workloads

experience with frequency scaling prevents the heuristic from engaging it. With seven instances of `stream` and one `aluadd`, frequency scaling can be engaged every other timeslice, and, with eight instances of `stream`, every timeslice.

Scalability with total number of tasks

In a next experiment, we vary not the percentage of memory-bound tasks in the workload, but the total number of tasks, keeping the ratio of memory-bound to compute-bound tasks constant at 1:1. Since with equal numbers of memory-bound and compute-bound tasks, frequency scaling is never invoked and the power consumption of the processor is almost the same for all scenarios, we only measured the runtimes of the benchmarks. Again, the runtime of `aluadd` does not vary between standard Linux scheduling and runqueue sorting.

Figure 5.18 shows the normalized runtime of `stream` for the two scheduling policies. Starting with eight instances total, sorted scheduling yields a significant benefit. With eight instances total, two tasks are available on each core and the scheduler has a choice which tasks to combine.

A smaller benefit is observable for six tasks. In this case, two cores are assigned one task each, while the remaining two cores are assigned two tasks each. Sorted scheduling can be applied to the latter cores, but not to the former.

The experiment shows that our scheduling policy scales reasonably well with the total number of tasks running on a chip, i.e, the benefits of our policy persist with an increasing number of tasks. We expect this property to hold also if the number of chips in the system is increased, since our co-scheduling policies are designed to be applied locally to chips (cf. Section 5.5.5).

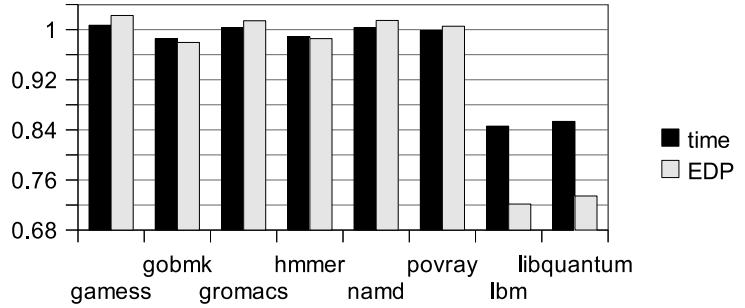


Figure 5.19: Effect of sorted co-scheduling on runtime and EDP—scenario with two memory-bound and six compute-bound benchmarks

5.8.4 Sorted co-scheduling

In the following, we evaluate sorted co-scheduling with the SPEC CPU 2006 benchmarks. We evaluate two scenarios: a scenario with two memory-bound and six compute-bound benchmarks, which is the combination that showed the most potential in our preliminary tests with the microbenchmarks, and a combination of four memory-bound and four compute-bound benchmarks.

For the scenario with two memory-bound tasks, we chose `lbm` and `libquantum` as two memory-bound benchmarks affected heavily by memory contention (cf. Section 5.3.4), and `gamess`, `gobmk`, `gromacs`, `hmmer`, `namd`, and `povray` as six compute-bound benchmarks hardly affected by contention.

Figure 5.19 shows the runtimes and EDPs of the benchmarks when sorted co-scheduling is applied, relative to standard Linux scheduling. While the compute-bound benchmarks' runtimes hardly change in comparison to standard Linux scheduling, the runtimes and EDPs of both memory-bound benchmarks decrease significantly, as could be expected from the preliminary experiments. For the memory-bound benchmarks, we achieve a reduction of runtime of 15% and a reduction of EDP of 28% on average.

The reduction of EDP observable for the memory-bound benchmarks stems almost completely from the reduction of runtime, since the power consumption stays almost the same with sorted co-scheduling. (Frequency scaling is never invoked, since at any point in time, three compute-bound benchmarks are running.)

Next, we ran four compute-bound benchmarks (`gamess`, `gromacs`, `hmmer`, `namd`) together with four memory-bound benchmarks (`lbm`, `libquantum`, `mcf`, `soplex`). Figure 5.20 shows the runtimes and EDPs of the benchmarks when sorted co-scheduling is applied, relative to standard Linux scheduling.

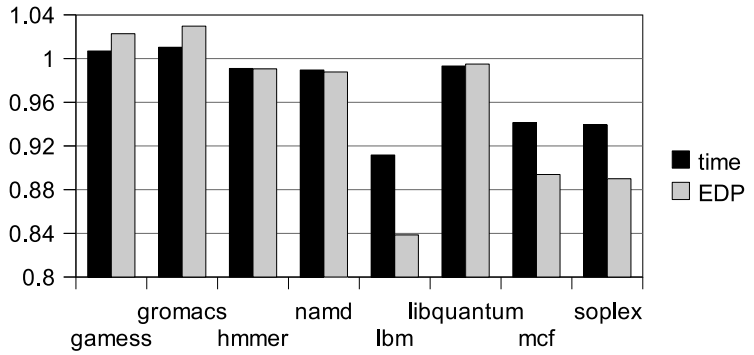


Figure 5.20: Effect of sorted co-scheduling on runtime and EDP—scenario with four memory-bound and four compute-bound benchmarks

Two things are worth noting: Firstly, the reduction of runtime and EDP for the memory-bound benchmarks is not as big as in the previous experiment, since in a scenario with four cores and a workload where half of the tasks are memory-bound, even with runqueue sorting, two memory-bound tasks have to share the memory bus at a time. (Note that the y-axis in Figure 5.20 has a different scale than the one in Figure 5.19).

Secondly, `libquantum`, though memory bound, shows only a slight reduction of runtime (and, as a consequence, EDP). This is also a result of the fact that, owing to the workload, we cannot avoid co-scheduling two memory-bound tasks at a time. These tasks have to share memory bandwidth. Depending on the tasks' memory access patterns, the memory controller can distribute bandwidth in an unfair way [MM07], which accounts for `libquantum` being at a disadvantage compared to the other memory-bound benchmarks.

Like in the previous scenario, the reduction of EDP observable for the memory-bound benchmarks stems almost completely from the reduction of runtime, since again, no frequency scaling is engaged.

5.8.5 Greedy co-scheduling

We ran the same scenarios as in the last section, but applied greedy co-scheduling instead of sorted co-scheduling. Since both co-scheduling policies do not influence the compute-bound benchmarks, we only compare the behavior of the memory-bound benchmarks under the two policies.

Since again, frequency scaling is never invoked for the given scenarios, the effects of our policies on energy and EDP only depend on runtime. Figure 5.8.5 com-

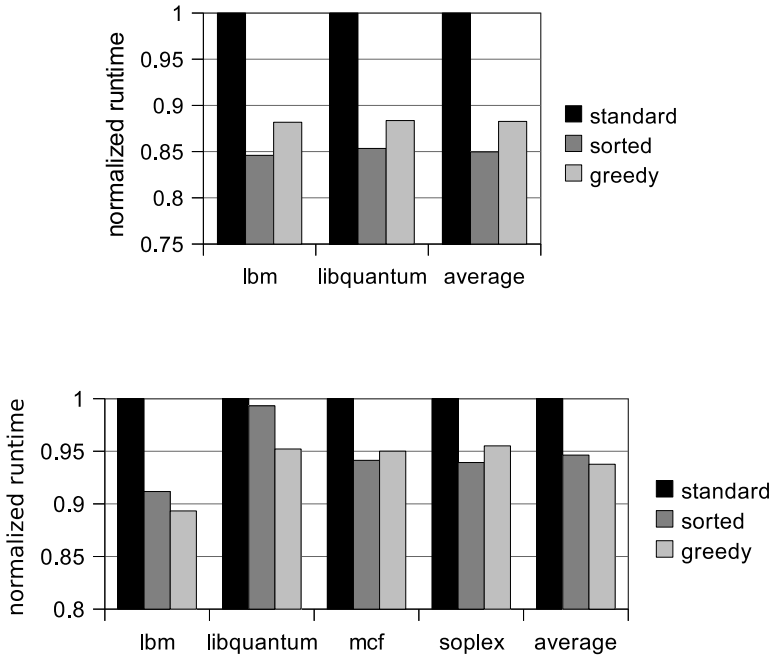


Figure 5.21: Comparison of greedy co-scheduling and sorted co-scheduling—scenario with two memory-bound benchmarks (top) and four memory-bound benchmarks (bottom)

pares the runtimes for the scenario with two memory-bound and six compute-bound benchmarks (top) and the scenario with four memory-bound and four compute-bound benchmarks (bottom). Two points are remarkable:

- When looking at the average, greedy co-scheduling achieves worse performance than sorted co-scheduling for the scenario with two memory-bound benchmarks, but slightly better performance for the scenario with four memory-bound benchmarks. The reason is that, depending on the situation, the policies choose different combinations of tasks to schedule. Since both policies are heuristics, neither one is guaranteed to select the best combination of tasks in all situations.
- Especially for the scenario with four memory-bound benchmarks, the two co-scheduling policies distribute performance improvements differently among the memory bound tasks, since they choose different combinations of tasks for co-scheduling.

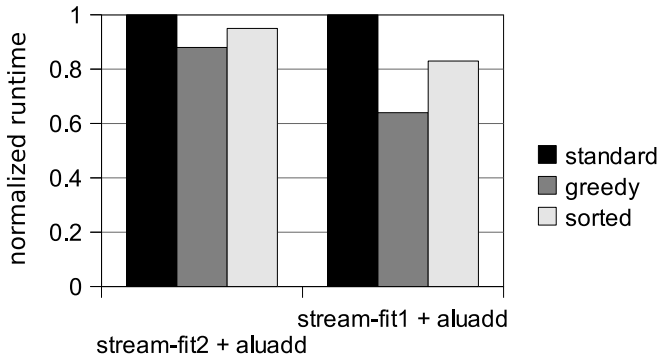


Figure 5.22: Comparison of greedy co-scheduling and sorted co-scheduling—normalized runtime of cache-bound tasks

While for the SPEC scenarios, overall, both co-scheduling policies showed comparable results, greedy co-scheduling is more advantageous than sorted co-scheduling for scenarios where memory is not the resource for which there is the heaviest contention. Greedy co-scheduling is more versatile owing to its uniform view on activity vectors, in contrast to sorted co-scheduling, which is fixed to one particular resource.

We ran four instances of the `aluadd` microbenchmark (no cache utilization) and four instances of `stream-fit2` (heavy cache utilization, cf. Section 5.3.4) on two cores sharing L2 cache. We also repeated the test with `stream-fit1` in lieu of `stream-fit2`. Figure 5.22 depicts the runtime of the cache-intensive tasks using standard Linux scheduling, sorted co-scheduling, and greedy co-scheduling, normalized to the runtime under standard Linux scheduling.

In both cases, greedy co-scheduling is more successful in finding an optimal schedule than sorted co-scheduling for memory bandwidth. The reason why sorted co-scheduling achieves an improvement at all is that whenever two cache-intensive tasks are scheduled together, the resulting cache misses cause utilization of memory bandwidth, which sorted scheduling can use to improve its schedule. However, the improved schedule results in less cache misses and memory bus activity, reducing sorted scheduling's future chances of finding the right schedule, and so forth.

Overall, the improvements that can be achieved for `stream-fit1` are much greater than those for `stream-fit2`, since `stream-fit1`'s working set fits into the cache only once, so co-scheduling two instances causes much more slowdown than this is the case with `stream-fit1`.

Scenario	**Benchmarks** (c = compute-bound, m = memory-bound)
1	gamess (c), gobmk (c), gromacs (c), hmmer (c), namd (c), povray (c), sjeng (c), tonto (c)
2	gamess (c), gromacs (c), hmmer (c), namd (c), lbm (m), libquantum (m), mcf (m), soplex (m)
3	hmmer (c), GemsFDTD (m), lbm (m), libquantum (m), mcf (m), milc (m), omnetpp (m), soplex (m)
4	GemsFDTD (m), lbm (m), libquantum (m), mcf (m), milc (m), omnetpp (m), soplex (m), sphinx3 (m)

Table 5.6: SPEC scenarios used for evaluating the frequency heuristic

5.8.6 Frequency heuristic

Our proposed frequency heuristic engages frequency scaling as a fallback to conserve energy in situations when co-scheduling cannot avoid resource contention. For the experiments with SPEC scenarios presented in the preceding sections, our frequency heuristic never invoked frequency scaling, because we selected scenarios containing enough compute-bound tasks for co-scheduling to be effective and reduce resource contention.

For evaluating the frequency heuristic, we choose four different SPEC scenarios. Scenario 1 contains only compute-bound benchmarks. Scenario 2 contains four compute-bound and four memory-bound benchmarks. For these two scenarios, frequency scaling should not be invoked; they are intended for revealing the overhead of our heuristic. Scenario 3 contains one compute-bound and seven memory-bound benchmarks, and scenario 4 contains only memory-bound benchmarks. Scenarios 3 and 4 represent the two configurations in which frequency scaling is beneficial, as indicated by our preliminary tests with the microbenchmarks (Section 5.8.3). Table 5.6 shows the individual SPEC benchmarks used for the scenarios.

We compare three configurations: Standard Linux scheduling, sorted co-scheduling, and sorted co-scheduling in combination with the frequency heuristic. Figure 5.23 depicts the average EDP for the SPEC benchmarks of our four scenarios, normalized to the EDP achieved by standard Linux.

For scenario 1, all three policies achieve the same EDP. The reason is that with eight compute-bound tasks, there is no contention that can be improved by co-scheduling, neither is there any opportunity to conserve energy by frequency scaling.

For scenario 2, there is contention that co-scheduling manages to reduce, since the workload contains four compute-bound and four memory-bound tasks. Engaging frequency scaling, however, is not beneficial because of the four compute-bound tasks. Therefore, the frequency heuristic yields no additional benefits over co-scheduling.

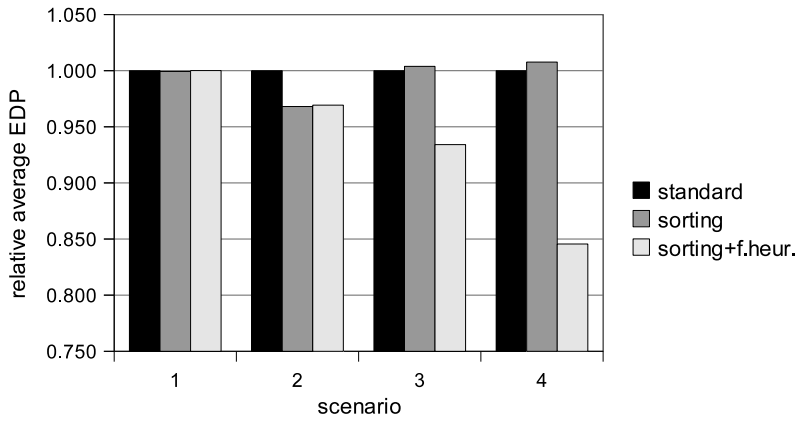


Figure 5.23: Effect of frequency heuristic for different SPEC scenarios—averaged normalized EDP of the benchmarks

Scenario 3 contains one compute-bound and seven memory-bound tasks. As a result, there is considerable memory contention, which co-scheduling cannot reduce since there are too few compute-bound tasks. In this situation, the heuristic can mitigate the effects of the memory contention on EDP by engaging frequency scaling. Yet, owing to the one compute-bound benchmark, frequency scaling cannot be invoked all of the time.

For scenario 4, which contains only memory-bound tasks, co-scheduling can, like in scenario 3, not avoid resource contention. The benefit of the frequency heuristic is even bigger for this scenario, since without any compute-bound tasks, the heuristic can lower the frequency all of the time.

As these experiments show, the frequency heuristic is able to detect situations in which frequency scaling can mitigate the waste of energy induced by resource contention without introducing additional energy inefficiency by prolonging the runtime of compute-bound tasks. The real advantage of our frequency heuristic over a static frequency setting appears for dynamic scenarios where the workload changes over time, and engaging frequency scaling is sometimes beneficial and sometimes not. We created such a workload by executing four instances of the memory-bound `mcf` in parallel on the four cores of our test chip, and after their termination, four instances of `hmmer`, and so on. Figure 5.24 shows the EDP achieved under the two static frequency settings of 2.4GHz and 1.6GHz, as well as under the frequency heuristic, relative to the EDP achieved at 2.4GHz. While `hmmer` runs more efficiently at 2.4GHz and `mcf` runs more efficiently at 1.6GHz, only the heuristic can choose the best frequency in each situation.

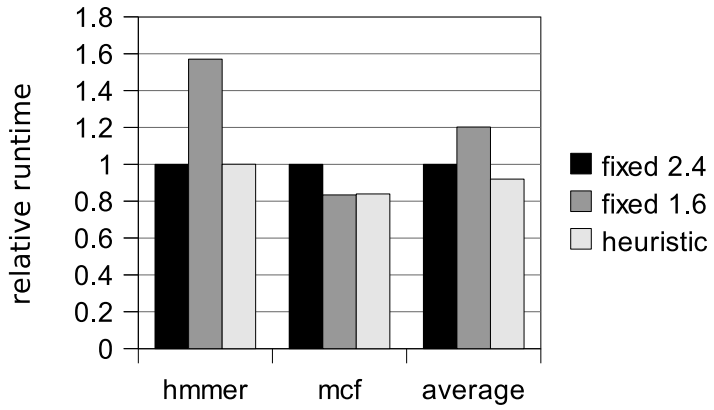


Figure 5.24: Effect of frequency heuristic for a dynamic workload

Overall, our experiments indicate that the frequency heuristic is capable of selecting the frequency that achieves or comes close to a minimal overall EDP, and is able to mitigate the effects of resource contention when our co-scheduling policies cannot avoid the contention owing to inauspicious workloads.

5.8.7 Application of co-scheduling to the NetBurst architecture

Since greedy co-scheduling is a policy that uses activity vectors in a uniform way, it can be applied without any adaption to any architecture that shares resources between processors, provided that there is an implementation of activity vectors representing the utilization of the resources. In particular, greedy scheduling should also lead to performance improvements for an SMT processor, since the logical processors share resources, and greedy scheduling avoids co-scheduling tasks that utilize the same resources.

For evaluating greedy co-scheduling on an SMT architecture, we applied it on top of our implementation of activity vectors for the Intel NetBurst architecture (see Chapter 4) and ran it on the IBM x440 system described in Section 4.6.

We ran the same scenario as described at the end of Section 4.6.5 (one physical processor enabled, Hyper-Threading enabled, six randomly selected SPEC benchmarks running in parallel). On average, greedy co-scheduling yielded a reduction of the benchmarks' runtime of 4.5% compared to standard Linux scheduling. This result underlines the versatility of vector-based scheduling policies. The improvement achievable with greedy co-scheduling is better than what activity unbalancing as described in Section 4.4.3 achieves (3.3% average reduction of runtimes). The reason is that greedy co-scheduling is focused explicitly on avoiding resource contention, in con-

trast to activity unbalancing, which was designed for avoiding hotspots and provided performance improvements as a byproduct.

5.9 Summary

In this chapter, we have investigated how energy efficiency can be improved by resource-conscious scheduling. We have considered the aspect of contention for shared resources and the aspect of selecting an optimal processor frequency based on workload characteristics. We have found that scheduling for reducing contention and scheduling for optimal frequency selection are oppositional goals if multiple cores have to run at the same frequency owing to hardware constraints. An analysis of the Intel Core2 Quad processor has revealed that, on this platform, co-scheduling tasks that utilize mutually different resources is more important than co-scheduling tasks that share a common optimal frequency. Depending on the workload, frequency scaling can lead to energy savings, but combining tasks that run best at a certain frequency does not pay off if it leads to resource contention.

We have shown that the information provided by activity vectors is suitable to guide resource conscious co-scheduling of tasks. Based on activity vectors, we have designed two policies that avoid contention and, as a consequence, reduce the energy delay product. *Sorted co-scheduling* is tailored to avoid contention for a single resource and is applicable if one resource is known to be the main bottleneck. We found memory bandwidth to be such a critical resource on the Core2 Quad. *Greedy co-scheduling* is suited to avoid contention for multiple resources and is applicable if there is no single bottleneck or if the most critical resource is unknown. Both policies rely on tasks with different characteristics being present on each core. We accomplish such a heterogeneous distribution of tasks to cores with *vector balancing*, a modified load balancing policy.

We supplement our scheduling policies with a heuristic for frequency scaling that yields an additional reduction of EDP in situations when our co-scheduling policies cannot avoid memory contention because of an inauspicious workload that contains too many memory-bound tasks.

Our evaluation has shown that our scheduling policies are able to reduce EDP for mixed scenarios containing compute-bound and memory-bound tasks. If the workload contains to many memory-bound tasks and contention cannot be avoided by scheduling, our frequency heuristic mitigates the effects of contention and as a consequence reduces EDP.

Our research has shown that on today's processors, contention for memory bandwidth is of paramount importance. It is an open question whether memory bandwidth will remain the most critical resource for future processors and future applications. However, our proposed policies are flexible; sorted co-scheduling can easily be adapted to consider any single resource, and greedy co-scheduling is not fixed on

a particular resource by design. No matter what future processors will be like, for economical reasons, chip resources will always be limited, and shared resources must be assigned carefully in order to avoid contention. Vector-based co-scheduling offers a system-software approach that accomplishes this goal by steering the utilization of scarce resources in a coordinated fashion.

6 Conclusion

6.1 Recapitulation

Developments in processor technology over the past decades have introduced new challenges, the most prominent being thermal problems, the need for energy efficiency, and explicit on-chip parallelism.

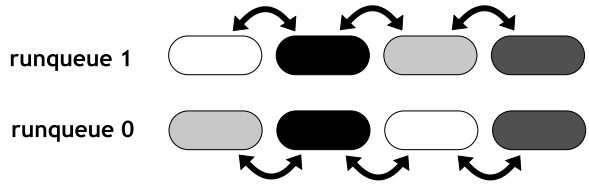
All three phenomena are strongly connected to the characteristics of individual tasks. Task characteristics determine chip temperature and temperature distribution, and task characteristics decide how efficiently mechanisms for conserving energy can be put to use. The characteristics of tasks executed in parallel influence the degree of contention for shared resources, affecting performance and energy efficiency.

Since the scheduler decides which tasks are executed at what time and in which combination, it is the scheduler who has control over these effects. It is our conviction that, on recent processors, scheduling in terms of optimal temperature, energy efficiency, and performance can only succeed if we provide the scheduler with information about individual tasks that goes beyond the task state that today's operating systems maintain.

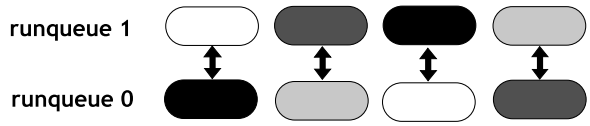
In this thesis, we have introduced the abstraction of *activity vectors*. An activity vector characterizes a task by the utilization of processor-related resources the task causes when executed. This allows the scheduler to assess the implications a particular schedule has on temperature, energy consumption, and performance.

We have proposed several scheduling policies based on activity vectors. For improving on-chip temperature distribution and avoiding hotspots, we schedule tasks that use different chip resources successively (*runqueue sorting*). On multithreaded processors, we achieve the same effect by scheduling tasks using different resources in parallel. We also propose a policy for distributing tasks to chips in a way that tasks using different resources are available on each chip to enable the other two policies.

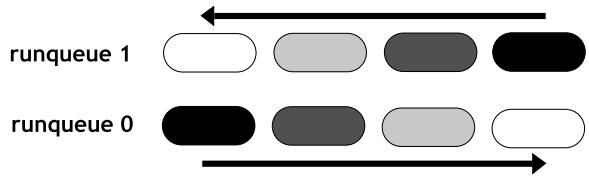
For improving energy efficiency and performance, we co-schedule tasks using different resources. We propose *sorted co-scheduling*, a specialized policy that is focused on a single bottleneck resource, and *greedy co-scheduling*, a generic policy that works if multiple resources cause contention. For situations in which contention for off-chip resources like memory bandwidth cannot be avoided, we propose to engage power saving mechanisms like frequency scaling to increase energy efficiency.



(a) Runqueue sorting for avoiding hotspots: tasks with complementary characteristics are scheduled in succession; combination of co-scheduled tasks is arbitrary



(b) Greedy co-scheduling: tasks with complementary characteristics are co-scheduled; temporal order of tasks is arbitrary



(c) Sorted co-scheduling: tasks are sorted according to utilization of a resource; implicit co-scheduling via reversed sorting direction of different queues

Figure 6.1: Comparison of thermal runqueue sorting, greedy co-scheduling, and sorted co-scheduling

6.2 Comparison of the Proposed Scheduling Policies

For comparison, Figure 6.1 depicts the policy of thermal runqueue sorting we introduced in Chapter 4 and the two co-scheduling policies introduced in Chapter 5 schematically.

Runqueue sorting for avoiding hotspots defines a temporal order on the tasks in a runqueue by selecting tasks with different characteristics for successive execution. Since co-scheduling is not performed, the combination of tasks running at a time is arbitrary. Greedy co-scheduling, on the other hand, applies the principle of selecting tasks with complementary characteristics across cores, but the temporal order in which the tasks run on each core is undefined. Finally, sorted co-scheduling defines a temporal order on the tasks in each runqueue, and implicitly co-schedules tasks by applying diametrical sorting directions.

Since thermal runqueue sorting does not depend on a specific combination of tasks running on different processors or cores at a time, whereas greedy co-scheduling does not depend on a specific order of the tasks executed in succession, we can combine the two policies for achieving both, reducing contention and improving temperature distribution. Sorted co-scheduling, on the other hand, defines its own order on the tasks in the runqueues; therefore, we cannot combine this co-scheduling policy with a policy for reducing hotspots. However, sorting by memory intensity is intended for systems where memory bandwidth is the most critical resource, for instance, CMP systems. For CMPs, overall energy efficiency is considered more important than localized heating [YSBZ05]: Localized heating occurs when there is a high activity in certain key structures, for example the register files of SMT processors that are utilized by multiple logical threads, whereas for CMPs, the main problem is that the aggregate power of several cores causes the entire package to heat up.

6.3 Achievements

Our thesis has introduced the concept of *task activity vectors* to represent a task's resource utilization and has shown that activity vectors provide a valuable metric for temperature and energy-aware, performance-oriented scheduling policies.

Activity vectors are an abstraction for representing resource utilization in general. While information about processor resource utilization has been used before to guide scheduling, the scheduling policies proposed in the literature have been hardwired to fixed architecture or micro-architecture specific metrics. To the best of our knowledge, our thesis is the first to abstract from the actual resources by introducing activity vectors whose components can represent any resource. This decouples the scheduling policy from the mechanism of determining resource utilization. As a consequence, activity vectors allow to formulate more general, versatile, and portable scheduling policies.

Furthermore, we have proposed temperature-aware scheduling and migration policies that make use of activity vectors. We have shown that scheduling guided by information about chip utilization can be used to attain a more balanced temperature distribution on the chip. Especially the idea of using the order in which tasks are scheduled for influencing temperature constitutes a new concept that is not to be found in the literature. Our experiments have shown that vector-based, temperature-aware scheduling succeeds in reducing hotspots.

We have also applied the concept of task activity vectors to achieve better energy efficiency via co-scheduling. To our knowledge, our work is the first to investigate co-scheduling in connection with both resource contention *and* frequency scaling at the same time. We have found that co-scheduling tasks in a way that avoids resource contention is more important for improving energy efficiency than co-scheduling tasks that profit from a common optimal chip frequency. Based on this knowledge, we have

proposed vector-based co-scheduling policies that reduce the energy delay product. Our experiments show that on a multicore chip, the energy delay product of memory-bound tasks can be reduced by up to 28% compared to standard Linux scheduling just by avoiding contention.

6.4 Limitations and Future Work

Obtaining activity vectors

Activity vectors, as proposed in our thesis, represent the utilization of processor-related resources. In our implementation, we have used performance monitoring counters to obtain information about resource utilization. While the approach of (ab)using performance monitoring counters for on-line task characterization has already proven viable in the past, it still has some drawbacks.

Performance monitoring counters are intended for monitoring events related to application performance, hence the selection of events made by the hardware designers does not necessarily serve the purpose of revealing the utilization of chip resources. As a result, for some resources, calculations involving multiple events must be applied [IM03]. Also, some performance events only allow a rough estimation of resource utilization. The limited number of suitable counters may necessitate multiplexing, which causes overhead for frequent reading and re-loading of performance counters and control registers. Multiplexing also requires a high timer interrupt frequency to deliver acceptable results. Last, if performance monitoring counters are used by the operating system for supporting scheduling, they are not available for their original purpose of performance analysis.

We have shown that information about resource utilization is a valuable information that can be used to improve temperature, performance, and energy efficiency. Therefore, we argue that it would be beneficial to include counters directly tailored to resource utilization in future hardware. This would eliminate the drawbacks mentioned above.

Vector-based scheduling

The main limitation of almost all of our proposed scheduling policies lies in their dependence on the workload. Our policies for improving temperature distribution or energy efficiency make use of the heterogeneity of the workload by controlling the combination of tasks that run simultaneously or successively on a processor. For workloads consisting only of tasks with similar characteristics (that means, tasks utilizing the same chip resources), our proposed scheduling policies are not beneficial. In addition, policies that change the order in which tasks are scheduled (e.g., runqueue sorting) are only applicable for runqueues consisting of multiple tasks.

While these conditions are certainly not present in every system, recent trends like virtualization and server consolidation combine more, potentially heterogeneous workloads in a single system. For future work, it would be beneficial to explore to what extent these techniques can be used in connection with vector-based scheduling. Especially the concept of virtual machine migration is interesting, since it allows to change the workload in a system.

In this thesis, we have deliberately neglected the aspect of multithreaded applications. The question what kind of characterization is most suitable and useful to describe the characteristics of multithreaded tasks (e.g., one activity vector for the entire task, one activity vector for each thread, or a combination of both) remains for future research. In addition, multithreaded tasks add further challenges for the design of vector-based scheduling policies. Next to the tasks' and threads' characteristics, relations between the threads, defined, for example, by communication or data sharing have to be considered for optimal scheduling.

The design of our scheduling policies has concentrated on non-interactive workloads. While we have not explored the space of interactive workloads, we believe that vector-based scheduling policies can be devised for such workloads, too. Owing to the volatile nature of interactive tasks, such scheduling policies should focus on the placement of newly starting or de-blocking tasks instead of arranging runqueues. Information about the characteristics of tasks running only for a short time could be provided using cached information from previous runs; we have described a comparable approach in previous work [Mer05].

An issue we have not explicitly addressed is the aspect of fairness. While with our policies, task characteristics do not influence the amount of processor time allotted to a task, changing the assignment of tasks to processors, as we do with our balancing policies, can have implications on fairness. For example, a task that, owing to its characteristics, is always scheduled in time-sharing fashion on the same CPU with a task with a large cache footprint, is penalized compared to a random assignment of tasks to processors. Similarly, while our co-scheduling policies are designed in a way that avoids resource contention, which is generally advantageous in terms of performance, situations in which co-scheduling penalizes a task with moderate resource demands by combining it with a resource intensive task are not precluded. Investigating the implications of vector-based scheduling policies on fairness is a topic for future work.

Interaction with future hardware

Future processors are likely to have built-in mechanisms and policies for avoiding hotspots. For instance, chips that feature spare resources and perform activity migration in hardware have been proposed [HBA03, SSH+03]. Another example are multithreaded chips that favor certain logical processors when allocating resources depending on the characteristics of the tasks they run and on the current temperature distribution [DM05].

We believe that both hardware and software should cooperate to attain the goal of effective temperature management best. If the hardware exposes information to the operating system about internal features like multiple register files or adaptive fetch policies, and about the actions actually taken to prevent thermal emergencies, the operating system can provide the circumstances in which the hardware's mechanisms are most effective, for example by migrating tasks between CPUs or ordering the tasks in the runqueues accordingly.

Another promising approach is allowing the software greater control over the hardware. Cazorla et al. [CKS+04] proposes to include a feature in SMT processors that allows the operating system to set the ratio at which the processor fetches instructions for the different logical processors. Cazorla et al. investigates this mechanism under the objective of providing quality of service. We believe that such a mechanism could also be used for regulating chip temperature based on information about the utilization of chip units as provided by activity vectors.

Our scheduling policies are based on the assumption that all processors are identical. Future multicore systems might contain cores that show different maximum frequencies and different power consumptions. This offers new possibilities and new challenges for scheduling [TT08]. A further modification we might see in future systems are cores that differ in their microarchitecture residing on one chip. In such systems, the assignment of tasks to cores has great implications on energy efficiency [CJ08]. Especially the issue of resource contention, which still persists for such heterogeneous architectures, has, to our knowledge, not been investigated yet.

List of Figures

List of Tables

Bibliography

[AB06] Jeff Andrews and Nick Baker. Xbox 360 system architecture. *IEEE Micro*, 26(2), 2006.

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP'97)*. ACM, October 1997.

[ACD06] James H. Anderson, John M. Calandrino, and Uma Maheswari C. Devi. Real-time scheduling on multicore platforms. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, April 2006.

[AHH89] Amant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), 1989.

[BB95] Thomas D. Burd and Robert W. Brodersen. Energy efficient CMOS microprocessor design. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*. IEEE Computer Society, 1995.

[Bel97a] Frank Bellosa. Follow-on scheduling: Using TLB information to reduce cache misses. In: Sixteenth Symposium on Operating Systems Principles (SOSP '97), Work in Progress Session, October 1997.

[Bel97b] Frank Bellosa. Process cruise control: Throttling memory access in a soft real-time environement. Technical Report TR-I4-97-2, University of Erlangen, Department of Computer Science, July 1997.

[BH04] Erik Berg and Erik Hagersten. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, March 2004.

[BM01] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*. IEEE Computer Society, January 2001.

[BP04] James R. Bulpin and Ian A. Pratt. Multiprogramming performance of the Pentium 4 with Hyper-Threading. In *Third Annual Workshop on Duplicating, Deconstruction and Debunking*, June 2004.

[BP05] James R. Bulpin and Ian A. Pratt. Hyper-threading aware process scheduling heuristics. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*. USENIX Association, April 2005.

[BPA08] Mohammad Banikazemi, Dan Poff, and Bulent Abali. PAM: a novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*. IEEE Computer Society, November 2008.

[BPJ+07] Sarah Bird, Aashish Phansalkar, Lizy K. John, Alex Mercas, and Rajeev Idukuru. Performance characterization of SPEC CPU benchmarks on Intel's Core microarchitecture based processor. In *SPEC Benchmark Workshop*, January 2007.

[BS96] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1), 1996.

[BWWK03] Frank Bellosa, Andreas Weissel, Martin Waitz, and Simon Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, September 2003.

[CCF+07] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Hendrik Hamann, Alan Weger, and Pradip Bose. Thermal-aware task scheduling at the system software level. In *Proceedings of the 2007 International Symposium on Low-Power Electronics and Design (ISLPED'07)*. ACM, 2007.

[CGKS05] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, February 2005.

[CJ08] Jian Chen and Lizy K. John. Energy-aware application scheduling on a heterogeneous multi-core system. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'08)*, September 2008.

[CKS+04] Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez, and Mateo Valero. Predictable performance in SMT processors. In *CF '04: Proceedings of the 1st conference on Computing frontiers*. ACM, April 2004.

[CMSB+08] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the Seventeenth Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, October 2008.

[CRW07] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Keith Whisnant. Temperature aware task scheduling in MPSoCs. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE'07)*. EDA Consortium, April 2007.

[CSP04] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 International Symposium on Low-Power Electronics and Design (ISLPED'04)*. ACM, August 2004.

[DM05] James Donald and Margaret Martonosi. Leveraging simultaneous multithreading for adaptive thermal control. In *Second Workshop on Temperature-Aware Computer Systems (TACS'05)*, June 2005.

[DM06] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. *SIGARCH Computer Architecture News*, 34(2), 2006.

[DR07] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proceedings of the 2007 International Symposium on Low-Power Electronics and Design (ISLPED'07)*. ACM, August 2007.

[EMGAD06] Ali El-Moursy, Rajeev Garg, David H. Albonesi, and Sandhya Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *20th IEEE International Parallel and Distributed Processing Symposium, 2006 (IPDPS 2006)*. IEEE Computer Society, April 2006.

[Fed06] Alexandra Fedorova. *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Harvard University, September 2006.

[FEL03] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Interaction of power-aware memory systems and processor voltage scaling. In *Proceedings of the Workshop on Power-Aware Computer Systems (PACS'03)*, December 2003.

[Fle01] Marc Fleischmann. Longrun power management. White Paper, Transmeta Corporation, January 2001.

[FPL+07] Vincent W. Freeh, Feng Pan, David K. Lowenthal, Nandini Kappiah, Rob Springer, Barry L. Rountree, and Mark E. Femal. Analyzing the energy-time tradeoff in high-performance computing applications. *IEEE Transactions on Parallel and Distributed Systems*, 18(6), 2007.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for finegrained synchronization. *Journal of Parallel and Distributed Computing*, 16(4), 1992.

[FSNS04] Alexandra Fedorova, Christopher Small, Daniel Nussbaum, and Margo Seltzer. Chip multithreading systems need a new operating system scheduler. In *EW11: Proceedings of the 11th ACM SIGOPS European workshop*. ACM, September 2004.

[FSSN05] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *ATEC '05: Proceedings of the annual USENIX Annual Technical Conference*. USENIX Association, April 2005.

[GBCH01] Stephen H. Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 2001. Q1 issue.

[GH96] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9), September 1996.

[GPV04] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. ACM, October 2004.

[HBA03] Seongmoo Heo, Kenneth Barr, and Krste Asanovi. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED'03)*. ACM, August 2003.

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4), 2006.

[HF04] Chung-Hsing Hsu and Wu-Chun Feng. Effective dynamic voltage scaling through CPU-boundedness detection. In *Proceedings of the Workshop on Power-Aware Computer Systems (PACS'04)*, December 2004.

[HF05] Chung-Hsing Hsu and Wu-Chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'05)*. IEEE Computer Society, November 2005.

[HIG94] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. In *IEEE Symposium on Low Power Electronics*. IEEE Computer Society, October 1994.

[HII+09] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. ACM, June 2009.

[HKK06] Yongkui Han, Israel Koren, and C. M. Krishna. Temptor: A lightweight runtime temperature monitoring tool using performance counters. In *Proceedings of the Third Workshop on Temperature-Aware Computer Systems (TACS'06)*, June 2006.

[HKS+07] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. *IEEE Transactions on Parallel and Distributed Systems*, 18(8), 2007.

[HM07] Sebastian Herbert and Diana 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)*. ACM, August 2007.

[HNR68] Peter E. Hart, Nils N. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), July 1968.

[HRIM06] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, September 2006.

[HSS+04] Wei Huang, Mircea R. Stan, Kevin Skadron, Karthik Sankaranarayanan, Shougata Ghosh, and Sivakumar Velusamy. Compact thermal modeling for temperature aware design. In *Proceedings of the 41st Design Automation Conference (DAC'04)*. ACM, September 2004.

[HWW02] Jim Hoskins, Bill Wilson, and Ray Winkel. *Exploring IBM EServer XSeries: The Instant Insider's Guide to IBM's Intel-Based Servers and Workstations*. Maximum Press, 2002.

[IBC+06] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MIRCO'06)*. IEEE Computer Society, December 2006.

[IM03] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MIRCO'03)*. IEEE Computer Society, December 2003.

[IM06] Canturk Isci and Margaret Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture (HPCA'06)*. IEEE Computer Society, February 2006.

[IMB05] Canturk Isci, Margaret Martonosi, and Alper Buyuktosunoglu. Long-term workload phases: Duration predictions and applications to DVFS. *IEEE Micro*, 25(5), September 2005.

[Int02] Intel Corporation. *Intel® Pentium® 4 Processor with 512-KB L2 Cache on 0.13 Micron Process Thermal Design Guidelines*, November 2002.

[Int06] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. 2006.

[JED08] JEDEC Solid State Technology Association. Failure mechanisms and models for semiconductor devices. *JEDEC Publication*, JEP122D, October 2008.

[Jon06] M. Tim Jones. Inside the Linux scheduler. *IBM Developer Works*, 2006.

[JSCT08] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, October 2008.

[JWP+05] Philo Juang, Qiang Wu, Li-Shiuan Peh, Margaret Martonosi, and Douglas W. Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In *Proceedings of the 2005 International Symposium on Low-Power Electronics and Design (ISLPED'05)*. ACM, August 2005.

[KCBB06] Eren Kursun, Chen-Yong Cher, Alper Buyuktosunoglu, and Pradip Bose. Investigating the effects of task scheduling on thermal behavior. In *Proceedings of the Third Workshop on Temperature-Aware Computer Systems (TACS'06)*, June 2006.

[KCCC08] Joonho Kong, Jinhang Choi, Lynn Choi, and Sung Woo Chung. Low-cost application-aware DVFS for multi-core architecture. In *International Conference on Convergence Information Technology (ICCIT'08)*. IEEE Computer Society, November 2008.

[KDG+04] Ramakrishna Kotla, Anirudh Devgan, Soraya Ghiasi, Tom Keller, and Freeman Rawson. Characterizing the impact of different memory-intensity levels. In *Proceedings of the Seventh IEEE International Workshop on Workload Characterization (WWC-7)*, October 2004.

[KGKR05] Ramakrishna Kotla, Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling processor voltage and frequency in server and cluster systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11*. IEEE Computer Society, April 2005.

[KGWB08] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture (HPCA'08)*, February 2008.

[KK06] Evangelos Koukis and Nectarios Koziris. Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of SMPs. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*. IEEE Computer Society, 2006.

[KSN07] Masaaki Kondo, Hiroshi Sasaki, and Hiroshi Nakamura. Improving fairness, throughput and energy-efficiency on a chip multiprocessor through DVFS. *SIGARCH Computer Architecture News*, 35(1), 2007.

[KSPJ06] Amit Kumar, Li Shang, Li-Shiuan Peh, and Niraj K. Jha. HybDTM: a coordinated hardware-software approach for dynamic thermal management. In *Proceedings of the 43rd Design Automation Conference (DAC'06)*. ACM, July 2006.

[KST04] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2), March 2004.

[KZT05] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. *SIGARCH Computer Architecture News*, 33(2), 2005.

[LCCF08] Wen-Yew Liang, Shih-Chang Chen, Yang-Lang Chang, and Jyh-Perng Fang. Memory-aware dynamic voltage and frequency prediction for portable devices. In *Proceedings of the Fourteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, August 2008.

[Lee06] Benjamin C. Lee. An architectural assessment of SPEC CPU benchmark relevance. Technical Report TR-02-06, Harvard University, January 2006.

[LFZE00] Alvin Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*. ACM, November 2000.

[LM06] Jian Li and José F. Martínez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture (HPCA'06)*. IEEE Computer Society, February 2006.

[LS05] Kyeong-Jae Lee and Kevin Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11*, April 2005.

[LSK04] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, February 2004.

[LVE00] Jochen Liedtke, Marcus Völp, and Kevin Elphinstone. Preliminary thoughts on memory-bus scheduling. In *EW 9: Proceedings of the 9th ACM SIGOPS European workshop*. ACM, September 2000.

[MAN05] Robert L. McGregor, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*. IEEE Computer Society, April 2005.

[MB06] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the First ACM SIGOPS EuroSys Conference*. ACM, April 2006.

[MB08a] Andreas Merkel and Frank Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower'08)*, December 2008.

[MB08b] Andreas Merkel and Frank Bellosa. Task activity vectors: A new metric for temperature-aware scheduling. In *Proceedings of the Third ACM SIGOPS EuroSys Conference*. ACM, March 2008.

[MBH+02] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Kofaty, J. Alan Miller, and Michael Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 2002. Q1 issue.

[McC95] John D. McCalpin. Sustainable memory bandwidth in current high performance computers, October 1995.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. ACM, March 2009.

[Mer05] Andreas Merkel. Balancing power consumption in multiprocessor systems. Diploma Thesis, Universität Karlsruhe (TH), System Architecture Group, September 2005.

[MM07] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *SS'07: Proceedings of 16th USENIX Security Symposium*. USENIX Association, August 2007.

[MS06] Pierre Michaud and Yiannakis Sazeides. Scheduling issues on thermally-constrained processors. Technical report, Institut de Recherche en Informatique et Systèmes Aléatoires, October 2006.

[Mud01] Trevor Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4), April 2001.

[NP02] Jun Nakajima and Venkatesh Pallipadi. Enhancements for Hyper-Threading technology in the operating system: seeking the optimal scheduling. In *WIESS'02: Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software*. USENIX Association, December 2002.

[NRM+06] Alon Naveh, Efraim Rotem, Avi Mendelson, Simcha Gochman, Rajshree Chabukswar, Karthik Krishnan, and Arun Kumar. Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*, 10(2), 2006.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Notices*, 31(9), 1996.

[Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*. IEEE Computer Society, October 1982.

[PELL00] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, May 2000.

[RHAH06] Efraim Rothem, Jim Hermerding, Cohen Aviad, and Cain Harel. Temperature measurement in the Intel Core Duo processor. In *Proceedings of the Twelfth International Workshop on Thermal Investigations of ICs (THERMINIC'06)*, August 2006.

[RLA07] Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. Thread scheduling for multi-core platforms. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. USENIX Association, May 2007.

[RLT06] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, September 2006.

[RWB09] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. ACM, June 2009.

[SABR04] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The case for lifetime reliability-aware microprocessors. *SIGARCH Computer Architecture News*, 32(2), 2004.

[SBB07] Joseph Sharkey, Alper Buyuktosunoglu, and Pradip Bose. Evaluating design tradeoffs in on-chip power management for CMPs. In *Proceedings of the 2007 International Symposium on Low-Power Electronics and Design (ISLPED'07)*. ACM, August 2007.

[SDR02] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, February 2002.

[SF91] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. *SIGPLAN Notices*, 26(4), 1991.

[SL93] Mark Steven Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), 1993.

[SLSPH09] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for OS-level power management. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, March 2009.

[SPH07] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Accurate online prediction of processor and memory energy usage under voltage

scaling. In *Proceedings of the Seventh ACM International Conference on Embedded Software (EMSOFT'07)*. ACM, October 2007.

[SPM07] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Process scheduling challenges in the era of multi-core processors. *Intel Technology Journal*, 11(4), 2007.

[SSH+03] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'03)*. ACM, June 2003.

[ST00] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-IX: Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*. ACM, November 2000.

[ST07] Kyriakos Stavrou and Pedro Trancoso. Thermal-aware scheduling for future chip multiprocessors. *EURASIP Journal on Embedded Systems*, 2007(1), 2007.

[SvdLPH07] David C. Snowdon, Godfrey van der Linden, Stefan M. Petters, and Gernot Heiser. Accurate run-time prediction of performance degradation under frequency scaling. In *3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2007.

[SW95] Patrick Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, April 1995.

[TAS07] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, March 2007.

[TE94] Radhika Thekkath and Susan J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*. IEEE Computer Society, April 1994.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*. IEEE Computer Society, June 1995.

[TT08] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, June 2008.

[VKT06] Matthew De Vuyst, Rakesh Kumar, and Dean M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE Computer Society, April 2006.

[VWWL00] Ram Viswanath, Vijay Wakharkar, Abhay Watwe, and Vassou Lebonheur. Thermal performance challenges from silicon to systems. *Intel Technology Journal*, 2000. Q3 issue.

[WA08] Jonathan A. Winter and David H. Albonesi. Addressing thermal nonuniformity in SMT workloads. *ACM Transactions on Architecture and Code Optimimizations*, 5(1), 2008.

[WB02] Andreas Weissel and Frank Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'02)*. ACM, October 2002.

[Wec06] Ofri Wechsler. *Inside Intel Core Microarchitecture*. Intel Corporation, 2006.

[WM08] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'08)*. IEEE Computer Society, September 2008.

[YC01] Lian-Tuu Yeh and Richard C. Chu. *Thermal Management of Microelectronic Equipment*. American Society of Mechanical Engineers, 2001.

[YSBZ05] Li Yingmin, Kevin Skadron, David Brooks, and Hu Zhigang. Performance, energy, and thermal considerations for SMT and CMP architectures. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture (HPCA'05)*. IEEE Computer Society, February 2005.

[ZDFS07] Xiao Zhang, Sandhya Dwarkadas, Girts Folkmanis, and Kai Shen. Processor hardware counter statistics as a first-class system resource. In *HOTOS'07: Proceedings of the 11th USENIX workshop on hot topics in operating systems*. USENIX Association, May 2007.

[ZII+07] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, September 2007.

A The SPEC CPU 2006 Benchmarks

For the evaluation of our scheduling policies, we use workloads composed of SPEC CPU 2006 benchmarks [Hen06]. SPEC CPU is a benchmark suite designed by the Standard Performance Evaluation Corporation (SPEC) and, according to the description provided by SPEC, is supposed "to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications". The benchmarks fall into the two broad categories of integer benchmarks and floating point benchmarks.

Though designed to compare the performance of CPU hardware, because of its wide coverage of application domains, SPEC CPU is suitable to evaluate operating system policies. This appendix gives a brief overview of the benchmarks.

Integer benchmarks

name	**programming language**	**description**
perlbench	C	PERL programming language
bzip2	C	compression
gcc	C	C compiler
mcf	C	combinatorial optimization
gobmk	C	artificial intelligence: go
hmmer	C	search gene sequence
sjeng	C	artificial intelligence: chess
libquantum	C	physics: quantum computing
h264ref	C	video compression
omnetpp	C++	discrete event simulation
astar	C++	path-finding algorithms
xalancbmk	C++	XML processing

Floating point benchmarks

name	programming language	description
bwaves	Fortran	fluid dynamics
gamess	Fortran	quantum chemistry
milc	C	physics: quantum chromodynamics
zeusmp	Fortran	physics: CFD
gromacs	C/Fortran	biochemistry: molecular dynamics
cactusADM	C/Fortran	physics: general relativity
leslie3d	Fortran	fluid dynamics
namd	C++	biology: molecular dynamics
dealII	C++	finite element analysis
soplex	C++	linear programming, optimization
povray	C++	image ray-tracing
calculix	C/Fortran	structural mechanics
GemsFDTD	Fortran	computational electromagnetics
tonto	Fortran	quantum chemistry
lbm	C	fluid dynamics
wrf	C/Fortran	weather prediction
sphinx3	C	speech recognition

B Deutschsprachige Kurzfassung

Moderne Mikroprozessoren – eine Herausforderung für die Ablaufplanung

In den vergangenen Jahrzehnten kam es zu einem ständigen Anstieg der Integrationsdichte und der Leistungsaufnahme von Mikroprozessoren. Dies führte zu einer Reihe von Problemen, mit denen wir uns heute konfrontiert sehen.

Aufgrund der gestiegenen Leistungsaufnahme pro Fläche ist die Abfuhr der in den Schaltkreisen freigesetzten Wärme zum Problem geworden. Es muss beträchtlicher Aufwand betrieben werden, um ein Überhitzen des Prozessors zu verhindern. Thermische Probleme, aber auch steigende Energiekosten sowie die zunehmende Verbreitung von mobilen Systemen mit begrenztem Energievorrat führen dazu, dass Energieeffizienz im Bereich der Mikroprozessoren zunehmend an Bedeutung gewinnt.

Für die heutigen Prozessoren sind Steigerungen der Prozessortaktfrequenz und der Prozessorkomplexität nicht mehr wirtschaftlich. Statt dessen finden wir in zunehmendem Maße expliziten Parallelismus in Form mehrerer Ausführungseinheiten auf einem Chip. Damit verbunden sind Abhängigkeiten zwischen den Ausführungseinheiten, die zum Beispiel durch gemeinsam genutzte Ressourcen oder gemeinsame Energieverwaltung entstehen.

Alle drei Aspekte – Temperatur, Energieeffizienz und Abhängigkeiten zwischen Ausführungseinheiten – weisen einen starken Bezug zur vom Prozessor ausgeführten Anwendung (*Task*) auf. Die Charakteristiken der ausgeführten Anwendungen bestimmen, wo auf dem Chip Wärme freigesetzt wird, wie effizient Maßnahmen zur Energieverwaltung wie das Skalieren der Frequenz sind und in welchem Maße sich Ausführungseinheiten durch die nebenläufige Nutzung gemeinsamer Ressourcen wie zum Beispiel des Speicherbuses gegenseitig beeinflussen.

Aus diesem Grunde fällt dem Ablaufplaner als der Komponente eines Betriebssystems, die entscheidet, zu welchem Zeitpunkt und in welcher Kombination welche Anwendungen ausgeführt werden, eine zentrale Rolle im Hinblick auf die oben genannten Probleme zu. In heutigen Betriebssystemen verwendete Strategien zur Ablaufplanung werden dieser Rolle nicht gerecht, da sie die Charakteristiken der von ihnen eingeplanten Anwendungen weder kennen, noch berücksichtigen und so suboptimale Entscheidungen treffen.

Task-Aktivitätsvektoren und vektorbasiertes Einplanen

Diese Arbeit führt das Konzept der Task-Aktivitätsvektoren zur Charakterisierung von Anwendungen ein. Ein Aktivitätsvektor ist Teil des Laufzeitkontextes einer Anwendung und charakterisiert die Anwendung über die von ihr verursachten Ressourcennutzung. Dabei entspricht jede Komponente des Vektors einer bestimmten Ressource und nimmt Werte entsprechend des Nutzungsgrades der Ressource an. Betrachtet werden in der Arbeit Prozessorressourcen wie arithmetisch-logische Einheiten, Gleitkommaeinheiten oder Caches, sowie der Speicherbus.

Der in dieser Arbeit vorgeschlagene Mechanismus ermittelt die Auslastung der Prozessorressourcen zur Laufzeit unter Verwendung von Ereigniszählern. Diese sind spezielle Prozessorregister, welche prozessorinterne Ereignisse zu zählen vermögen und ursprünglich zur Leistungsanalyse und -optimierung eingeführt wurden. So ist es möglich, den Auslastungsgrad der verschiedenen Ressourcen während der Ausführung einer Anwendung zu ermitteln und damit den Aktivitätsvektor der Anwendung zu bestimmen.

Aufbauend auf Aktivitätsvektoren schlägt diese Arbeit mehrere vektorbasierte Einplanstrategien vor. Diese nutzen die durch die Aktivitätsvektoren bereitgestellten Informationen, um eine ausgeglichenere Temperaturverteilung auf dem Chip bzw. eine Verringerung der Konkurrenz um gemeinsam genutzte Ressourcen – und damit höhere Performanz und gleichzeitig bessere Energieeffizienz – zu erreichen.

Zur Vermeidung von Hotspots, besonders heißen Stellen auf dem Chip, plant die entsprechende Strategie Anwendungen nacheinander ein, die jeweils verschiedene funktionale Einheiten des Prozessors nutzen. So tritt an keiner Stelle des Prozessors permanente Aktivität auf und die den funktionalen Einheiten entsprechenden Schaltkreise können in den Perioden der Inaktivität abkühlen.

Zur energieeffizienten Nutzung gemeinsamer Ressourcen plant die entsprechende Strategie Anwendungen dergestalt auf verschiedenen Ausführungseinheiten (d.h. Prozessorkernen oder Hardware-Kontrollfäden) ein, dass gleichzeitig ausgeführte Anwendungen verschiedene Ressourcen nutzen. So kann vermieden werden, dass Ausführungseinheiten auf die Freigabe einer schon belegten Ressource warten und Energie umsetzen, ohne dass die ausgeführte Anwendung Fortschritt machte.

Beide genannten Strategien erreichen ihr Ziel durch gezielte Sortierung von prozessorlokalen Listen laufbereiter Anwendungen. Die Strategien werden ergänzt durch eine Migrationsstrategie, welche die Anwendungen so auf die Prozessoren verteilt, dass auf jedem Prozessor Anwendungen mit unterschiedlichen Charakteristiken vorhanden sind.

Ergebnisse

Die Arbeit beinhaltet die Umsetzung des vorgeschlagenen Konzeptes der Task-Aktivitätsvektoren sowie der darauf aufbauenden Einplanstrategien für den Linux-Kern. Die Auswertung erfolgt auf einem IBM xSeries 440–Serversystem mit acht Intel P4-Xeon-Prozessoren sowie auf einem Desktopsystem mit einem Intel Core2 Quad–Prozessor.

Da sich die Temperaturverteilung in den Prozessoren mit den zur Verfügung stehenden Mitteln nicht feststellen lässt, erfolgt die Auswertung der vorgeschlagenen Strategien hinsichtlich der Temperaturverteilung mit dem an der University of Virginia entwickelten Temperatursimulator *HotSpot*. Die Auswirkungen der Strategien hinsichtlich der Performanz und der Energieeffizienz werden am tatsächlichen System gemessen.

Die Auswertung ergibt, dass sich durch vektorbasiertes Einplanen Hotspots signifikant reduzieren lassen. So lässt sich für ein Szenario aus SPEC CPU 2006–Benchmarks, für das unter dem Standard-Linux-Ablaufplaner während 25% der Laufzeit eine maximale Temperatur von mehr als 80°C auf dem Chip herrscht, dieser Anteil durch vektorbasiertes Einplanen auf 6% reduzieren. Dem gegenüber steht eine Erhöhung der Laufzeit im Bereich von 1%, hauptsächlich verursacht durch das häufige Auslesen der Ereigniszähler zum Bestimmen der Aktivitätsvektoren. Dieser Mehraufwand resultiert aus der beschränkten Zahl und Konfigurationsmöglichkeit der Ereigniszähler für die untersuchte Architektur und ließe sich durch geeignetere Zähler in der Hardware noch deutlich verringern.

Auch die Energieeffizienz lässt sich durch vektorbasiertes Einplanen deutlich verbessern. Für die untersuchte Architektur und die SPEC CPU 2006–Benchmarks stellt sich die Speicherbandbreite als die kritische Ressource heraus, deren Verwendung die größten Auswirkungen auf Performanz und Energieeffizienz hat. Für gemischte Szenarien bestehend aus speicherintensiven und rechenintensiven Benchmarks ergibt sich das größte Verbesserungspotential; durch vektorbasiertes Einplanen ist hier für die speicherintensiven Benchmarks eine Verbesserung des Produktes aus Laufzeit und aufgewandter Energie (*Energy Delay Product, EDP*) um bis zu 28% möglich.

Zusammenfassend lässt sich als Ergebnis der Arbeit festhalten, dass die Charakterisierung von Anwendungen über ihre Nutzung von Prozessorressourcen eine wertvolle Information darstellt, deren Berücksichtigung es entsprechenden Einplanstrategien erlaubt, die auf heutigen Prozessoren auftretenden Probleme der Temperatur, Energieeffizienz und der Konkurrenz um gemeinsame Ressourcen abzumildern.