Universität Karlsruhe (TH)
Research University · founded 1825

System Architecture Group
Department of Computer Science

Diploma Thesis

Impacts of Asymmetric Processor Speeds
on SMP Operating Systems

by

cand. inform.
Martin Röhricht

Supervisors:

Prof. Dr. Frank Bellosa

Dipl. Inf. Andreas Merkel

August 30, 2007

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, August 30, 2007

. . . .
Martin Röhricht

Acknowledgements

First of all, I would like to thank my advisors, Prof. Dr. Frank Bellosa and Andreas Merkel. Throughout the work on my thesis both provided me with inspiration and motivation. Without their help this thesis would not have been possible.

Thanks go to all members of the System Architecture Group in Karlsruhe for providing such a productive and friendly work environment. I would like to thank my colleagues Christoph Klee and Marco Schnurr for fruitful discussions. Both helped me with their expertise when I strayed onto the wrong track. David Talbott was a big help in linguistical questions.

Finally a very special thank you to my parents, Michael and Gudrun, as well as my girlfriend Karin. I am deeply grateful for all the selfless support and love they give me.

Contents

Abstract

Multiprocessor systems were the industry's response to an ever rising demand for more speed in conjunction with less heat and power dissipation. Instead of putting all effort into gaining more speed from a single-core system, hardware vendors focus today on building systems that consist of multiple cores. These systems are commonly referred to as Symmetric Multiprocessing (SMP) systems, where all processors or cores run with the same speed and build a homogeneous platform.

But instead of focusing on symmetric settings alone, in the future operating systems might have to deal with asymmetric, heterogeneous systems, which are built in a way that not every processor runs with the same speed.

This thesis examines what impacts such setups have on today's operating systems and develops strategies that promise a gain in terms of performance, stability and predictability.

We implemented different scheduling strategies for the Linux kernel that focus on different objectives, namely one approach that achieves the best minimal response times on average for all tasks in the system, another that achieves the highest possible throughput rate for the entire system, and finally one that takes each task's priority into account in order to guarantee the highest priority jobs are assigned to the fastest processors in the system.

Evaluations show that only an asymmetry-aware scheduler is able to make proper use of the big potential that is offered by a heterogeneous system. Our proposed solutions show performance gains of up to 50% compared to an unmodified scheduler. They show remarkable improvements in terms of performance stability and predictability.

1 Introduction

This thesis investigates the impacts of asymmetric processor speeds in a multiprocessor system and proposes operating system enhancements that are essentially designed to increase system performance. We will show that systems that are not aware of an underlying asymmetric topology perform poorly in general. We will introduce strategies which make proper use of the heterogeneity with different performance objectives. A comparison between a homogeneous and a heterogeneous setup confirms the theories by benchmark tests.

1.1 Homogeneous Multicore Architectures

Over the past decades processor technology evolved mainly by increasing frequency values to fulfill the consumer's needs. Architectural level changes such as sophisticated pipelining, out-of-order execution or branch prediction focused mainly on an improvement of *Instruction Level Parallelism*. Hyperthreading was later introduced to establish another logical processor by replicating some resources such as register sets and the interrupt controller but still sharing other resources among the logical processors like ALUs and caches. This technique is commonly referred to as Simultaneous Multithreading (SMT) and improved the system's performance with regard to *Thread Level Parallelism*.

In response to the ever increasing circuit density and heat dissipation of modern single-core systems such as Intel's Pentium IV processor with its netburst architecture, most semiconductor manufacturers focus today on developing multiprocessing or multi-core systems by including more than one CPU on a single physical package such that all processors are identical and share a common main memory. These Symmetric Multiprocessing Systems (SMP) offer a better performance per Watt balance, and provide a big potential for multithreaded applications. They allow the execution of multiple instruction streams simultaneously and can still be combined with SMT technology to further improve performance.

Chip Multiprocessing (CMP) is a special variant of SMP systems where a portion of on-die cache (e.g. L2 cache and above) is shared between the cores. Examples are dual-core processors such as Intel's Yonah implementation for a CMP system, and one of Intel's Pentium D processor families for a typical SMP system.

CMP setups can be especially useful when it comes to a reasonable utilization factor of a system. Memory intensive applications are known to underutilize a modern CPU because of the still increasing performance gap between the CPU and the memory subsystem. Multi-core systems can improve the overall utilization by the execution of more than one thread that would otherwise have had to wait some time for data from

memory. By increasing the number of cores and reducing the core's clock frequency the peak system performance will still increase and may help reducing the overall power consumption and heat dissipation.

We will use the terms *CPU*, *processor* and *core* interchangeably in this paper. The same convention applies to the terms *task*, *process* and *job* which should not be distinguished herein.

1.2 Heterogeneous Multicore Architectures

The above-mentioned multiprocessor systems are *homogeneous* in the sense that they provide the user equal performance per core and the same Instruction Set Architecture (ISA). This is also commonly referred to as Symmetric Multiprocessing (SMP). In order to apply rules that can be used to fulfill different user's needs the operating system should be aware of the underlying topology.

But multiprocessor systems do not necessarily need to be homogeneous. Provided that each processing unit uses the same ISA, it is possible to create systems where each processing unit executes these instruction sets at a different performance level. Using the same ISA on all cores is necessary to allow all processes in the system to execute on every available core. We will refer to such systems as being *heterogeneous* or *asymmetric*. Note that performance asymmetry contrasts to functional asymmetry where each processor entity is equipped with a different set of capabilities such that tasks must be matched with CPUs possessing the capabilities they need. In contrast to the commonly used term SMP we refer to heterogeneous systems also as Asymmetric Multicore Platforms, or simply AMP.

The basic idea of asymmetric multicore architectures was postulated some decades ago. Liu et al. published research on the theory of different scheduling algorithms for real-time systems in heterogeneous environments in the early 1970's [LY74]. Recent studies have not necessarily been limited to real-time, but have focused more on the case of single-ISA heterogeneous architectures, as described by Ghiasi et al. [GG03, GKR05, KDG+04]. In response to the lack of appropriate heterogeneous hardware prototypes, most of today's work is based on simulations. Balakrishnan et al. [BRUL05] investigated the impacts of performance asymmetry on a wide range of commercial applications using real hardware prototypes.

An asymmetric multicore architecture that is composed of cores with different computational capabilities may provide a number of low performance simple cores which provide an efficient use of computational power in terms of power dissipation and given thermal budgets for highly parallel applications, and a specific amount of complex cores that, while inefficient, suit the needs of computational intensive single threaded applications best.

The quantity of cores of each type and their organization on one chip may vary based on the particular system requirements. The designers of heterogeneous architectures focus on different objectives. Examples are business constraints such as the actual costs of particular cores and their availability. The designer can also be bound to specific power

and thermal budgets that should not be exceeded by the die or a given aggregated area for all cores. Thus there are multiple options and constraints that need to be taken into consideration which leads to a broad diversity of setups. Figure 1.1 illustrates two possible compositions which may stem from the constraint of available die area.
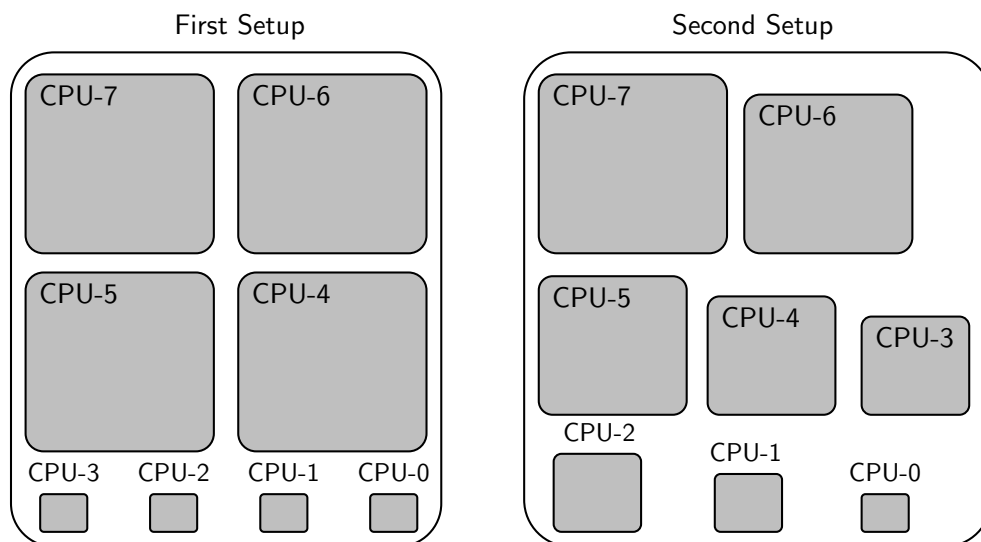

Figure 1.1: Two possible setups of asymmetric processor speeds on systems with eight CPUs. Setup one consists of two groups where each of the group's CPUs run with the same speed. Setup two is composed of eight CPUs running with eight different speeds.

A setup composed of multiple cores where each runs at a different speed suits the needs of any given application in response to its particular runtime characteristics especially well. A multiprocessor system composed of two groups with two different performance levels is easier to implement and would perhaps be even more cost-efficient by limiting the levels of complexity. This is a reasonable tradeoff between business constraints and technological demands.

But even today we face multiple systems that can be adjusted in speed at runtime. Frequency scaling is one well-known example of a technique that offers the operating system the opportunity to change the processor's frequency according to system demands and leaves the user uncertain about any predictability of the upcoming system's performance. With the rise of multicore platforms, different cores can be adjusted to different speeds at runtime which already introduces the notion of an asymmetric setup. For example, Intel's Santa Rosa design allows one of two cores residing on the same chip to operate temporarily at a higher frequency level.

1.3 Motivation

The design of heterogeneous systems breaks a long-standing assumption among developers, that all CPUs in a multiprocessor system should provide equal performance. An SMP-aware operating system is primarily concerned with an even distribution of tasks among all available processors in order to achieve a best possible utilization of all cores to provide all tasks in the system with the best possible performance. This implies that runnable tasks would be assigned to processors with the least load so that no processor stays idle while another one has to serve two tasks at a time, and that all tasks in the system are provided the same amount of CPU power (with respect to the number of tasks running on one CPU).

Today's general purpose operating systems go into architectural details and use sophisticated implementations to further improve performance and achieve power savings if necessary. Enhancements cover the distinction of different forms of multiprocessor systems, such as Hyperthreading, SMP, CMP, or NUMA systems. A profound knowledge of the operating system about the underlying hardware components, their state and their topology is very important as it leads to more beneficial decisions.

Hyperthreading systems, for example, introduce new logical processors. These share some basic components such as execution units and caches. Whenever hyper-threading is active in multiprocessor systems consisting of multiple physical processors, the scheduler should be aware of both dimensions and handle them differently.

Such details are already addressed in today's operating systems like GNU/Linux which offers special performance or power saving policies. Assume two tasks are to be assigned by the scheduler to two of these four logical CPUs. It would be rather detrimental in terms of performance, if the scheduler decided to assign both tasks to one physical CPU, as this is likely to result in cache thrashing where each task overwrites parts of the shared cache lines whenever it is active. But in terms of power savings it might be beneficial to leave one physical processor idle, if this CPU can be put in a sleep state. If a third task were to enter the system it would be advantageous to put the one with the highest priority on a separate CPU.

We want to investigate the impacts of performance-asymmetric multicore organizations on today's operating systems and focus on some key points of interest.

1.3.1 Predictability of a workload's performance

Using an asymmetric setup may lead to varying workload performance if the operating system is not asymmetry-aware. If the operating system scheduler does not assign tasks to processors based on the actual computational power of a particular CPU but rather on an even distribution of tasks among all CPUs, we face unpredictable behaviour in terms of workload performance. This is not acceptable for some commercial servers that must meet certain performance guarantees.

The system's throughput is a valuable metric for performance predictability as well as each single task's response time. An asymmetry-aware system should predict the workloads performance quantitatively and qualitatively—e. g. higher priority tasks should

predictably show better performance metrics than their lower priority counterparts.

A system that takes asymmetric processor speeds into account should behave equally given equal circumstances and workloads. Results should be reproducible and not left to chance.

1.3.2 Improvement of the system's performance

We must declare and model ways to increase the total system's performance. Performance can be measured in various ways, two of them being the total system's throughput rate and each task's response time. We will talk about those two metrics in Section 2.2.

If an operating system is not aware of different processor speeds in a multiprocessor system, it may assign tasks to slow CPUs, while leaving the faster CPUs idle. Within an SMP system we do not necessarily have to distinguish between response time and throughput, as an even distribution of tasks among all processors leads to the best possible performance results in both respects. In this case, we will always avoid leaving one CPU entity in the system idle, which maximizes the throughput rate. On the other hand we will only assign more than one task to a particular CPU, if all other CPUs are loaded, too. As all CPUs run equally fast in an SMP system, this automatically leads to best response times per task.

But if we face an asymmetric setup, faster cores should be primarily picked for assignment, and the distribution of tasks differs depending on whether we want to maximize either throughput or each task's response time.

1.4 Asymmetry Aware Scheduling

An operating system must be aware of heterogeneity in a system's setup; otherwise performance degradation can be observed. This affects the process scheduler of the operating system, which is responsible for the system's process assignment and load balancing between the cores during runtime.

Assume an asymmetric setup of CPUs in terms of computational power and a scheduler that is not aware of this situation. The scheduler would primarily be concerned about an even distribution of tasks among processors to yield an increased throughput rate. It will not consider assigning tasks to particular processors to increase their response times as it treats all CPUs in the entire system equally. This results in suboptimal scheduling decisions most of the time such that the faster cores are not primarily loaded with tasks prior to any slower core. Furthermore, even bigger differences between cores, that would allow assigning proportionally more tasks to the faster one to achieve best possible response times, would not be recognized at all.

Yet, heterogeneity offers a lot of convenient options that can be utilized by the operating system. These options range from maximized throughput to decreased power consumption. The latter may be applied by exploiting different program phases and the resulting instructions per cycle variations. Memory intensive applications may be served equally well or just with minor performance degradation on slower processors due to the

performance gap between the processor and the memory subsystem. This strategy may be applied by processors that offer frequency/voltage scaling such that any core in a multiprocessor system can be run at different speed which leads to power savings.

1.5 Structure

The rest of this thesis is organized as follows. Chapter 2 discusses the background and related work in the field of dynamic thread assignment on multiprocessor architectures and research that was done in the past on heterogeneous systems. Chapter 3 covers details of the theories developed to take advantage of asymmetry. Chapter 4 discusses our implementation which is based on the open source GNU/Linux operating system. Chapter 5 investigates the strategies applied by providing experimental results before Chapter 6 finally concludes.

2 Background and Related Work

We want to provide a brief overview of existing types of asymmetric multiprocessing systems and introduce the general operation principles of multiprocessor scheduling. This chapter concludes with an analysis of related work in the field of heterogeneous multiprocessors.

2.1 Different Types of Asymmetry

Asymmetric multiprocessing systems are already present in different flavours; some setups are permanently asymmetric whereas others are only temporary for a certain period during runtime.

2.1.1 NUMA – Non-uniform Memory Access

The NUMA model is a shared memory computer architecture for multiprocessor systems where each processor can access its own pool of memory directly, and can furthermore access memory pools of other processors via an interconnection. The access times are closely related to the memory location, i. e. a processor can access its local memory much faster than memory that is managed by a different processor.

The entire NUMA system consists of multiple individual systems, called nodes. How these nodes are connected to one another differs, e. g. via a dedicated processor-to-processor interconnection or through an external front-side bus and memory controller hub. In contrast to SMP where all memory accesses are posted to the same shared memory bus, NUMA limits the number of CPUs on any one memory bus. The various nodes are then connected by means of a high speed interconnect which surpasses the scalability limits of the SMP architecture. Hence NUMA systems allow for a greater number of CPUs competing for access to the shared memory bus.

Today's operating systems take these architectural characteristics into account and form logical groups of nodes to build a representation of the underlying topology. The distance between any two nodes may serve as a direct performance indicator. Hops between nodes, latency and bandwidth are popular metrics to determine the distance. Migrating a task from one node to another is a typical case in which the specific distance costs have a direct influence on the scheduler's decision.

2.1.2 Clock Modulation

A common technique to prevent a processor from overheating is to throttle the processor by periodically disabling the clock signal for a certain amount of time, hence the name *clock modulation*. The amount of throttling is most often specified as a percentage—a value of 50% states that every second clock cycle is discarded and thus results in a halved processor performance.

Intel implemented clock modulation by integrating an additional special circuit, the thermal monitor, on the Pentium 4 core. This monitor frequently checks the current CPU's temperature against a certain critical value. In case the temperature exceeds this critical value, the thermal monitor sends a PROCHOT# signal (from "processor hot") to enable the Thermal Control Circuit (TCC) which determines how many clock cycles should be omitted in order to reduce CPU heat dissipation, and which modulates the frequency accordingly.

Figure 2.1 shows the basic working principal. Whenever the PROCHOT# signal is active, the internal clock duty cycle control excludes some of the clock cycles periodically. This results in lowered heat dissipation but also reduces the CPU's performance as a lower resulting frequency is sent to the ALUs.
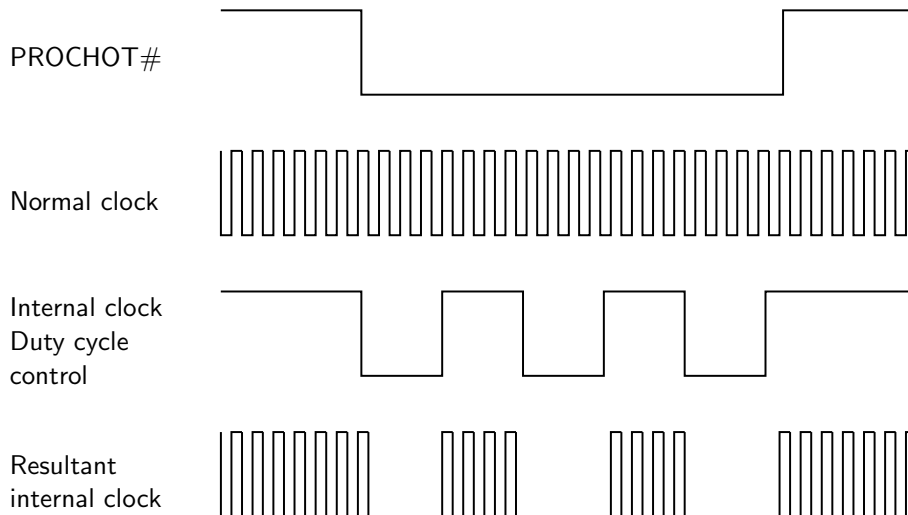
Figure 2.1: The modulation of the clock signal sent to the CPU.

Though the actual power dissipation can be reduced nearly linearly, the primary reason it is widely used in today's systems is the possibility to react quickly in case of emergency when the internal heat dissipation exceeds certain limits.

Given that in a multiprocessing system clock modulation can be used individually for each core which results in an asymmetric setup.

2.1.3 Frequency/Voltage Scaling

Variation of the clock's frequency affects the actual power consumption of any CMOS microprocessor nearly linearly. In modern processors with up to 600 million transistors,

additionally, the leakage power has a major influence on the total energy consumption. For example, the Pentium 4 processor dispenses about 40 Watt in idle mode.

Leakage power can be drastically reduced by lowering the supply voltage. Dynamic Voltage Scaling (DVS) is a technique to reduce power consumption of a chip during runtime. It exploits the fact that the dissipated power P of CMOS circuits is strongly dependent on the core voltage V and the clock frequency f such that

$$P \propto fV^2$$

This yields the possibility of a proportionally quadratic reduction of power consumption and is thus widely used as the primary method for power reduction.

Systems that operate with a reduced or varying frequency because of voltage/frequency scaling are already present in numerous of today's multiprocessor systems and can therefore build asymmetric setups.

2.1.4 Asymmetric Hardware on one Node

In contrast to all aforementioned types of asymmetric setups, we may face hardware in the future that is composed of heterogeneous cores by design. This is not necessarily limited to specialized cores like mathematical or graphical units but could refer to a system consisting of multiple general purpose cores with more than one single speed.

Such systems could be composed in a way such that the accumulated die area of all cores does not extend a certain limit. Furthermore the different cores would not necessarily need to implement the same ISA.

A small number of complex cores offers good serial performance whereas a large number of simple cores performs particularly well on a workload with a high degree of parallelism. Simple cores are especially useful in regard to an efficient use of transistors, as they are able to meet strict power and thermal constraints. Complex cores help to improve performance and time constraints for some workloads.

2.2 Multiprocessor Scheduling

The process scheduler is a major entity in a time-sharing operating system since it periodically decides which process is chosen to run next on which CPU and how long this process is allowed to run without interruption according to its timeslice. Switching from one task to another in a very short time frame gives the impression of simultaneous execution.

The operating system aims towards the best possible system utilization, for which it defines a set of rules, the scheduling policy, describing how and when processes are assigned to processors. But the measurement of computer productivity may lead to conflicting objectives such as achieving the highest *throughput rate* versus a better *response time*.

Throughput is defined to be the amount of work that can be performed by a computer in a given time period. In order to achieve the best throughput rates a system should try to execute as many tasks as possible and avoid idle cycles whenever possible. For multiprocessor systems, that results in the fact that no processor should be left idle while any one task in the system is ready to be executed but is forced to wait for another (running) task to be preempted.

The response time deals with the performance of individual tasks or a set of processes and hence is measured rather by the responsiveness of any given task. It is defined to be the time from which a user issues an event until the event is processed and the response is sent back to the user. Figure 2.2 illustrates the definition of the response time.
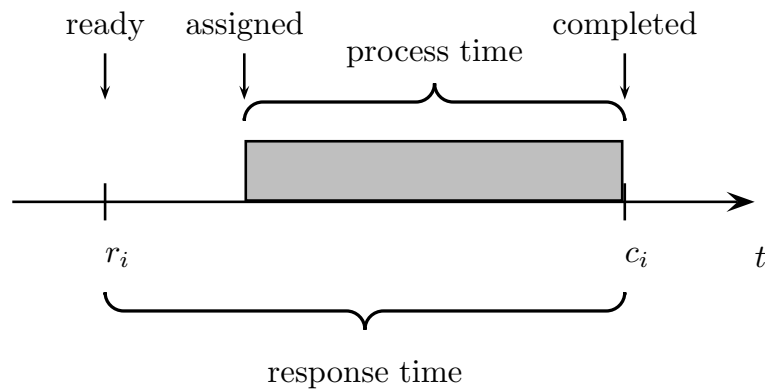

Figure 2.2: The response time of a task t_i is measured from the stimulus at r_i until the user gets feedback at time c_i.

The term completion time (c_i) in Figure 2.2 is neither limited to the completion of the entire task nor the one of a particular timeslice, but rather to the completion of an event that the user waits for from its stimulus at time r_i.

With regard to asymmetric systems a fast processor needs less time to process an event than its slower counterparts. This may lead to a situation in which it is advantageous to leave single nodes idle that perform worse, while stronger nodes in the system serve more than one task. Some applications are considered to be more interactive than others, that is the application responds directly to user activity and thus it is critical to process interactive tasks quickly, otherwise the user will perceive the system to be unresponsive. The same applies to soft real-time applications that deal with individual deadlines. For example video decoders have to discard frames consistently if the processor is not fast enough.

Operating systems may implement a large set of diverse scheduling strategies. A typical strategy in a time-sharing environment is called Round Robin. This policy grants all processes access to the processor consecutively for a limited period of time, a so-called timeslice. The Round Robin policy uses a queue for runnable processes. After the expiration of a given timeslice the active process is preempted and reinserted at the end of the queue, while the head of the queue is assigned to the processor. In Linux, the

length of a given task's timeslice is directly related to its priority, i.e. a higher priority results in a longer timeslice.

Multiprocessor operating systems are able to assign tasks to any given processor and are usually capable of migrating them amongst processors dynamically at runtime. The kernel code is likewise executed on each CPU simultaneously and locking mechanisms need to ensure data integrity. Therefore each CPU runs an own instance of the scheduler, and the data structure a scheduler is primarily working on to organize the scheduling of sets of tasks, is usually called a *runqueue*. Every task belongs to one runqueue only at any given point in time. The allocation of tasks to particular runqueues may lead to load imbalances in case of unequal runqueue lengths. The more tasks are assigned to a CPU the less CPU time can be provided for each single task. Therefore sophisticated schedulers aim towards balancing runqueue lengths as much as possible.

Furthermore the scheduler has to deal with process priorities. The Linux kernel for example uses 140 different priorities—a higher priority is reflected by a lower *priority value* of a given task. Priorities 1 to 99 are reserved for real-time tasks, priorities 100 to 139 can also be used by non real-time applications. As mentioned earlier, process priorities have a direct influence on the process's timeslice in Linux. A priority of 100 results in a timeslice length of 800 ms, the default priority of 120 will grant the process a timeslice of 100 ms and the lowest priority in Linux with 139 will result in a base time quantum of only 5 ms. Timeslice lengths do not increase in a linear fashion but are rather calculated by a special formula. Besides this *static* priority of a task, the scheduler honors or penalizes tasks periodically during runtime with a *dynamic* priority, based on their runtime behaviour and average sleep time. That is interactive tasks and tasks that have been denied the use of the CPU for a long time get a bonus, whereas very CPU intensive tasks will get a decreased dynamic priority.

Time-sharing operating systems allow processes to be preempted whenever their timeslices expire. The responsiveness of a system is directly related to the length of the task's timeslices—while short timeslices would indeed provide all tasks the opportunity to execute their code more frequently, it may lead to a workload where no process makes real progress at all, due to a significant runtime overhead caused by process switches that become excessively high.

2.2.1 Hierarchical Load Balancing

Load imbalances occur frequently in a highly dynamic system and the scheduler tries to solve these events by migrating tasks from highly loaded runqueues to lightly loaded ones. But task migrations come at a cost, because even though a task was interrupted by the scheduler, it is likely that the CPU's cache is filled with data from this particular task when the task is reassigned later. In this case, the task is said to be cache hot and migrating it would result in performance penalties.

But not only the cache is affected by task migrations. Tasks in NUMA systems even possess a node-affinity which adds a new dimension to the affinity problem. To prevent migrations from being penalized in terms of performance, we must try to avoid transfers across node boundaries and prefer migrations between CPUs belonging to the same

node. Once a task is migrated from one node to another one, we face the problem that we would have to migrate the corresponding memory to the task's new location or the task is forced to perform inter-node accesses.

We face multiple different dimensions concerning the actual system's underlying topology, and in order to allow the scheduler to make optimal decisions the scheduler needs to be topology-aware. The Linux kernel divides the system's entities into scheduling domains. Figure 2.3 shows an example of an eight way multiprocessor which consists of two NUMA-nodes where each of which is composed of four single processors.
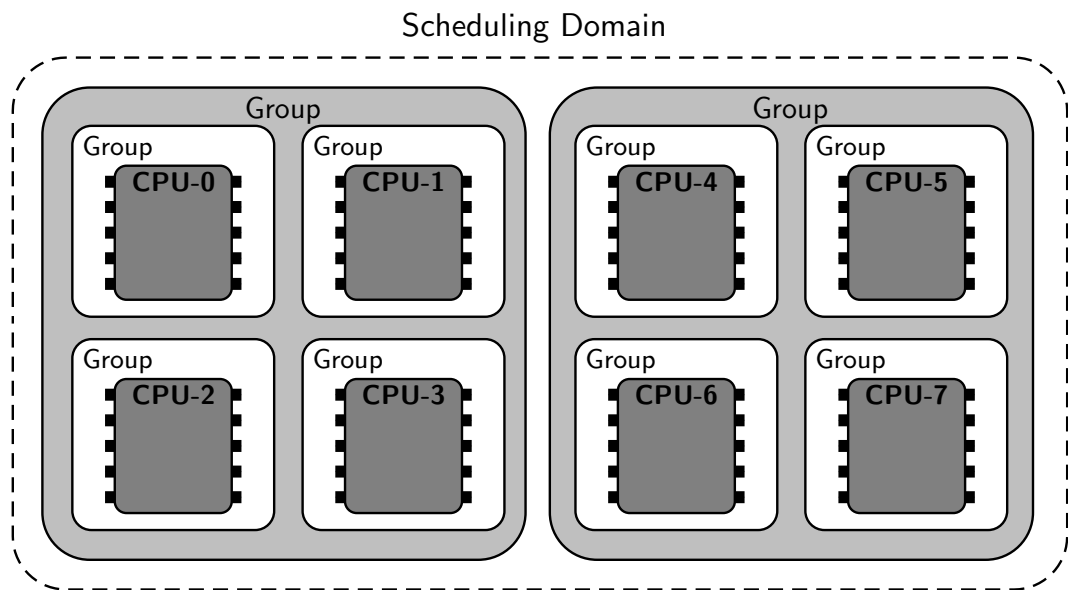

Figure 2.3: Scheduling Domain and different levels of Groups as seen by the Linux Scheduler in a NUMA-System with two nodes each consisting of four CPUs.

Scheduling domains are set up hierarchically, i. e. the top-most domain, which usually contains all CPUs in the system includes children scheduling domains which are composed of a subset of CPUs. The subsets themself are organized in groups. Generally speaking the higher the difference between levels of domains during migration, the more cost intensive the balancing operations therein are. Therefore the Linux scheduler aims towards balanced scheduling domains on each level.

We will talk details relevant to the implementation of the Linux kernel process scheduler in section 4.1. However, it is important to note that groups may be composed of multiple physical processors, but are also used as the lowest entity within a scheduling domain, such that a single CPU is represented by a corresponding group as well.

16

2.3 Related Work

2.3.1 Real-time Related Approaches

Calandrino et al. [CBL$^+$07] discuss possible ways of soft real-time scheduling on asymmetric multicore platforms using a GNU/Linux operating system. The authors apply a global view on the different performance characteristics of each core and assign a task to the slowest core that still satisfies its real-time requirements.

Liu et al. [LY74] already examined the case of optimal scheduling algorithms for preemptive schedules with minimal completion times as well as non-preemptive schedules with minimal mean flow times in 1974. Their approach is based on the theory of a standard processor and processors of speed b being b times as fast as a standard processor. Furthermore, their study requires an *a priori* knowledge of each task's execution time in relation to the standard processor.

Dealing primarily with real-time systems, Piel et al. [PMSD04] extended the Linux kernel with more real-time and heterogeneity awareness by classifying the system into real-time and non real-time CPUs such that real-time tasks are admitted to the former ones and non real-time tasks are only allowed to run on real-time CPUs if they are not currently utilized. The scheduler's pull-strategy had to be changed into a more active push-strategy.

2.3.2 Finding the optimal Setup

Different workloads perform quite differently on heterogeneous setups, depending on the workload's characteristic as well as the underlying architecture's topology. Kumar, Tullsen and Jouppi [KTJ06] examine the ideal setup of heterogeneous cores when running multiple workloads. Their simulation is based on Alpha processors and takes many micro-architectural structures into account, such as cache sizes or the size of the register sets. In conclusion the study states that the best heterogeneous systems are not constructed of cores that are commonly used as general purpose processors or cores that are used in today's homogeneous SMP systems, but rather designed in such a way that each individual core maps a specific class of applications with common characteristics.

Ghiasi and Grunwald [GG03] analyzed ways to determine the most suitable solution for an asymmetric dual core system based on specific design goals such as minimized power, energy and leakage as well as the number of fan-out-of-four inverter delays and finally overall throughput. The Wattch simulator was chosen to evaluate single-ISA, heterogeneous processors of the same x86 family, namely the Pentium Pro, Pentium II, Pentium III and Pentium IV processors.

2.3.3 Designs based on Simulations

Single-ISA heterogeneous multi-core architectures are evaluated in different studies by Kumar et al. [KFJ$^+$03, KTR$^+$04] in respect to workload performance and potential power reduction. These initial proposals use different existing cores of the Alpha processor

family placed on one single die. The work is based on the methodology of replacing bigger complex cores by multiple but smaller simple cores that occupy the same die area. Based on simulators, the studies showed a big potential for *performance improvements* and *reducible power consumption.* The study proved that static thread assignments even with best knowledge of the specific task's characteristic is inferior to a dynamic task and thread migration strategy.

A very promising evaluation was done by Ghiasi et al. [GKR05, KDG+04] where a heterogeneous system with identical ISAs was explored with regard to scheduling tasks with different execution characteristics. Specifically the *memory intensity* of executing tasks and ways to schedule them in such a way that *maximum power savings* can be accomplished by minimal performance loss was the primary focus of this work. The authors implemented a task-to-frequency scheduler for the Linux kernel that calculates the ideal frequency for an executing task online and schedules these tasks based on fixed processor frequencies and their current capacities. The paper shows that big power savings are possible by taking the memory intensity of workloads into account on a heterogeneous system.

A very thorough analysis on the impact of performance asymmetry was carried out by Balakrishnan et al. [BRUL05] where commercial applications were used to quantify the impact of heterogeneity on a system's performance variance and the application's stability. The authors examined the effects of making either the operating system scheduler asymmetry-aware or making the application aware of the underlying topology. The paper concentrated on predictability, scalability and ways to eliminate negative impacts. The study proved that asymmetry can adversely effect the system's performance if the application already works with fixed assumptions, coming to the conclusion that some applications need more help than the operating system can provide, and that the performance of an asymmetric system is much better than in a system where all cores run at a slower speed due to better serial performance of the fast cores.

A different approach was taken by Bender and Rabin [BR00]. The Cilk platform provides an ANSI C language extension to develop multithreaded parallel programs. Its runtime system is already delivered with a separate scheduler under the assumption that every processor is only aware of its own status. The author's goals were to design a representation of a parallel program as an acyclic graph and minimize its makespan. This problem is considered NP-hard and scheduling must occur offline. The existing scheduler was extended such that faster processors try to obtain more tasks than slower ones.

Grochowski et al. [GRSW04] investigate the potential of power savings on an architectural level by observing the fact that processor technology evolution has been characterized by a falling performance-to-power-consumption-ratio. They conclude that a combination of voltage/frequency scaling and asymmetric cores is a promising approach.

Based on the simulation of two Alpha processors running with different speeds but using the same ISA, Becchi and Crowley [BC06] find a correlation between IPC variation and program phases and conclude that some learning mechanisms need to be established as a heterogeneous system only shows significant benefit in the case of topology-aware software and may even affect the overall performance adversely in case of unaware system

software. How program phases can be discovered and exploited is discussed in detail by Sherwood et al. [SPH+03].

Asymmetric cluster chip multiprocessor (ACCMP) architectures are explored in emulations by Morad et al. [MWK04, MWK+06]. The designers base their experimentation on the same ISA and a given constant die area on which cores of different sizes are placed. They claim that a better performance can be achieved by dividing the die area into asymmetric cores rather than into symmetric ones. The scheduler is made responsible for reasonable thread assigning strategies based either upon hints supplied by the thread or by sampling the runtime properties of the thread.

In contrast to all aforementioned published papers of this section whose work is based on simulations of heterogeneous multiprocessors, we worked on a real multiprocessor system with cores running at different speeds. Section 5.1 discusses the test setup and working environment in more detail.

2.4 Methodology

The way we refer to a heterogeneous setup is closely related to the model introduced by Liu and Yang [LY74]. A so-called standard processor is considered to run at speed 1. Processors running at a different speed are considered to run at speed b if they are b times as fast as the standard processor. Without loss of generality, $b > 1$ for all non-standard processors, that is the standard processor is the slowest one in our system.

A heterogeneous multiprocessor system consists of n_1 processors of speed b_1, n_2 processors of speed b_2, ..., n_k processors of speed b_k. We refer to a system in general as

$$S = (n_1, n_2, \ldots, n_k; b_1, b_2, \ldots, b_k).$$

This states that we have N processors in total with $N = n_1 + n_2 + \cdots + n_k$ and an individual processor is referred to as P_i with $i = 1, \ldots, N$. This way the class of processors with speed b_1 consists of $P_1, P_2, \ldots, P_{n_1}$, the class of processors with speed b_2 consists of $P_{n_1+1}, P_{n_1+2}, \ldots, P_{n_1+n_2}$ and so on.

Using this model, we can express a given system's setup easily by its number of processors and their corresponding speed relative to the slowest processor in the system. For example, $S = (2, 6; 4, 1)$ states that we have eight processors in our system, two running with a speed that is four times as fast as the speed of each of the six remaining processors. Such a system may be composed of two processors running with 2.2GHz and six processors running with 550MHz, respectively.

3 Asymmetry-Aware Scheduling

The process scheduler of an operating system is the system's entity that can respond to an asymmetric multiprocessor the best, as it perpetually assigns tasks to processors at runtime.

The key point of interest is to find a suitable strategy on how tasks should be assigned to processors. A scheduling strategy aims at one or more objectives, such as a maximized performance outcome, and may be based on different aspects such as the task's priorities, possible migration costs or specific runtime characteristics. Some objectives may be mutually exclusive under specific circumstances like a maximized throughput rate versus an optimal per task response time. Section 2.2 discusses this aspect in greater detail.

We propose three different scheduling strategies that take asymmetric processor speeds in a multiprocessor environment into account and assign tasks to processors under specific targets.

3.1 Best Response Time Scheduling

We mentioned earlier in this paper that the response time of a single task may serve as an appropriate metric for performance measurements. The response time deals with the responsiveness of a task and is therefore defined as the time a system or functional unit takes to react to a given input. Consequently, the response time is bound to the system's processing speed.

For example, consider the input is to decode a video frame or to parse a text segment. The response time is the time from the beginning of the transaction to its first reaction, i.e. until the first video frame is decoded and sent back to the user. The response time is a self-contained per task metric and does not deal with other tasks or the state of any of the processors.

Most often the aim is to minimize the response time for a single task. Having more than one task in the system, we typically aim towards a minimal response time on average, that is the accumulated response times of each single task divided by the number of tasks (the arithmetic mean). If there are more tasks in the system than processors, we will eventually face the situation in which more than one task needs to be assigned to one single processor. The response time is therefore affected by

(a) the actual speed of the processor the task runs on,

(b) the number of tasks, the processor has to serve concurrently, and

(c) the timeslices of each particular task on a processor.

With regard to the actual speed of a processor and the number of tasks that share one such CPU, we will define the *effective CPU power* to be the speed that a single task receives by dividing the actual CPU power by the number of running tasks. For example by having a processor clocked at 900 MHz and three active tasks running on it, each task receives an effective CPU power of 300 MHz.

A SMP operating system will always try to distribute tasks evenly among all available processors, such that the difference of the number of tasks between any two processors in the system never exceeds one. For example, by having a four-way multiprocessor system the first four tasks are each assigned to different processors. While the fifth task can generally be assigned to any of the four processors, the sixth task is bound to one of the remaining three processors that are loaded with one task only at that point in time. This provides a maximized throughput, because as many cores as possible are utilized. Furthermore this yields the best possible response times on average for all tasks, because all processors run with the same speed and there is no way to distribute the tasks in a different way in order to achieve higher effective CPU power values.

But dealing with asymmetric setups may lead to situations in which it is desirable to assign more than one task to a specific processor while another one is still idle, in order to guarantee minimal response times on average. Consider two processors of different speed, CPU-0 running with 1 000 MHz and CPU-1 running with only 200 MHz. Generally speaking this leads to a situation in which CPU-0 is five times faster than CPU-1. Assume the response time for a given task is based on the decoding of a video frame and this job can be completed in one second on the faster processor, CPU-0, but would require five seconds on CPU-1. If two instances of this task were active and we assigned each task to one single CPU, we would end with an average response time of three seconds. If we assigned both tasks to the faster CPU however, we would be served with an average response time of only two seconds, as both tasks have to share the CPU and get only 500 MHz of effective CPU power which translates to two seconds for this job.

3.1.1 Approach on a timeslices/speed ratio

Our proposed strategy is based on the approach that faster cores should handle proportionally more tasks than their slower counterparts. We want to reduce the response time and offer as many tasks as possible the opportunity to achieve a performance gain from the faster cores in the system.

As mentioned earlier in this chapter, the timeslices of each task differ according to their priorities and have an impact on the actual response time, i. e. by having two tasks being executed on one CPU, the task with the bigger timeslice will be served with a better response time. Therefore we concentrate on the ratio of a task's timeslice and any given speed of a present processor in the system. We refer to this ratio as the *load* which is incurred by a particular task i on a particular CPU j:

$$\text{load}_{ij} = \frac{\text{timeslice}(i)}{\text{CPU power}(j)}$$

According to our methodology detailed in Section 2.4 we assume to have k processors, $n_1, \ldots, n_k$, running at speeds $b_1, \ldots, b_k$ proportional to the slowest processor in the system, and n tasks $t_1, \ldots, t_n$ with corresponding timeslice lengths $\tau_1, \ldots, \tau_n$. Timeslices are not necessarily of equal length and processor frequencies may vary.

When a scheduler decision takes place, we calculate the vector v_i for a given task t_i by dividing the specific task's timeslice by each single processor speed, such that

$$v_i = (\tau_i/b_1, \tau_i/b_2, \ldots, \tau_i/b_k) \tag{3.1}$$

Once we know how much load a specific task with a given timeslice would cause on each processor, we calculate which of the available processors would best fit the needs of this task, if it were to be scheduled on it, such that the system-wide load is minimized. That is, we look for the CPU which will end up with the least accumulated load after a possible task assignment.

Each CPU in the system is assigned a specific number of tasks and each task increases the load of this CPU according to its timeslice length as seen in formula 3.1. In order to attain the current total load L_j of CPU j, we need to accumulate the load of all tasks on this CPU separately.

$$L_j = \sum_{i=1}^{m} \frac{\tau_i}{b_j} = \frac{\tau_1 + \tau_2 + \cdots + \tau_m}{bj} \tag{3.2}$$

All k CPUs are now represented by a system-wide load vector ℓ which holds all individual CPU's total loads:

$$\ell = (L_1, \ldots, L_k) \tag{3.3}$$

The next step consists in a summation of the task i's vector v_i and the CPU's load vector ℓ. The outcome would now indicate the resulting load of all CPUs in case the task would have been assigned to each of those processors:

$$\ell + v_i = \left(L_1 + \frac{\tau_i}{b_1}, \ldots, L_j + \frac{\tau_i}{b_j}, \ldots, L_k + \frac{\tau_i}{b_k} \right) \tag{3.4}$$

We focus on the lowest value within this vector and the corresponding CPU is chosen to be the one where the task should be assigned to, in order to guarantee best response times on average. That is, we are looking for the index j that holds the minimum value of $\ell + v_i$. If e_j denotes the unit vector for position j, we schedule task t_i on CPU-j and add the value of $(\tau_i/b_j) \cdot e_j$ to ℓ.

This model ensures that high performance cores are utilized proportionally to their speed and the speed of all the other cores in the system. That leads eventually to workload setups in which a faster core is loaded with more than one task, while slower

cores may stay idle. But as we already discussed at the beginning of this section, this is sufficient whenever one of two cores is substantially slower and tasks are better served by sharing one fast core than running exclusively on a slower one.

However this strategy will not necessarily maximize the total system's throughput rate as the computing power of any idle core should be utilized prior to assigning more than one task to a processor in this case, regardless of how fast it runs compared to the slower core.

3.1.2 Example

Given four CPUs with corresponding frequencies:

$$f_1 = 250 \text{ MHz}, \quad f_2 = 250 \text{ MHz}, \quad f_3 = 500 \text{ MHz}, \quad \text{and } f_4 = 1\,000 \text{ MHz}.$$

Following our methodology we would face the current setup of

$$(2, 1, 1; 1, 2, 4)$$

We assume five tasks enter the system in this particular order:

- t_1 with $\tau_1 = 150$ ms,

- t_2 with $\tau_2 = 200$ ms,

- t_3 with $\tau_3 = 150$ ms,

- t_4 with $\tau_4 = 50$ ms,

- t_5 with $\tau_5 = 10$ ms,

- t_6 with $\tau_6 = 300$ ms.

Then the calculation for the task-to-processor assignments will be as follows:

Step 1: Current load vector $\ell = (0, 0, 0, 0)$; Task t_1 with $\tau_1 = 150$ ms enters the system.
Resulting temporary vector $v_1 = (150, 150, 75, 37.5)$.
$\rightsquigarrow L = (150, 150, 75, \mathbf{37.5}), \quad j = 4$.
Schedule task t_1 on CPU-4.

Step 2: $\ell = (0, 0, 0, 37.5)$. Task t_2 with $\tau_2 = 200$ ms enters the system.
$v_2 = (200, 200, 100, 50)$.
$\rightsquigarrow L = (200, 200, 100, \mathbf{87.5}), \quad j = 4$.
Schedule task t_2 on CPU-4.

Step 3: $\ell = (0, 0, 0, 87.5)$. Task t_3 with $\tau_3 = 150$ ms enters the system.
$v_3 = (150, 150, 75, 37.5)$.
$\rightsquigarrow L = (150, 150, \mathbf{75}, 87.5), \quad j = 3$.
Schedule task t_3 on CPU-3.

Step 4: $\ell = (0, 0, 75, 87.5)$. Task t_4 with $\tau_4 = 50$ ms enters the system.
$v_4 = (50, 50, 25, 12.5)$.
$\rightsquigarrow L = (50, \mathbf{50}, 100, 100) \quad j = 2$.
Schedule task t_4 on CPU-2.

Step 5: $\ell = (0, 50, 75, 87.5)$. Task t_5 with $\tau_5 = 10$ ms enters the system.
$v_5 = (10, 10, 5, 2.5)$.
$\rightsquigarrow L = (\mathbf{10}, 60, 80, 90), \quad j = 1$.
Schedule task t_5 on CPU-1.

Step 6: $\ell = (10, 50, 75, 87.5)$. Task t_6 with $\tau_6 = 300$ ms enters the system.
$v_6 = (300, 300, 150, 75)$.
$\rightsquigarrow L = (310, 350, 225, \mathbf{162.5}), \quad j = 4$.
Schedule task t_6 on CPU-4.

This example illustrates the instantaneous task-to-processor assignment whenever a task enters the system. We see from the example above, that CPU-4 handles three tasks, whereas the other three processors handle only one each. This is based on valid scheduling decisions in terms of the best response times on average, as CPU-4 is substantially faster than the remaining CPUs in the system.

3.1.3 Load Balancing

Until now, we have covered only the distribution of newly started tasks to different processors which can be derived directly from the above model. However, this model is rather static and does not take dynamic characteristics of a system into consideration, that is tasks may enter or leave the runqueues at any given point in time. Therefore, we expand our strategy to perform load balancing on all processors periodically.

Load balancing is performed frequently by today's general purpose SMP-aware operating systems. The Linux kernel scheduler for example compares runqueues of physical processors against each other in the order of every other 100 milliseconds. The focus lies on the most even distribution possible of running tasks among the available processors. However, since these operating systems are currently not asymmetry-aware, they assign processes to CPUs according to the number of running tasks in the system, no matter how fast one core operates. Scheduling decisions are thereby likely to be suboptimal in terms of performance considerations.

We need to provide a load balancing mechanism that is invoked periodically by the scheduler and which detects any imbalances between processors. But an imbalance is not defined to be based on the number of running tasks per CPU in this regard. Rather

we speak about an imbalance if the current distribution of tasks among the processors does not yield the best possible results in form of mean response times.

Once the scheduler detects an imbalance, it should resolve it automatically by migrating tasks from one processor to another so that the average response time of all running tasks in the system increases. In order to perform load balancing in this context, all cores with their corresponding load values and processing power need to be considered and compared to one another.

As described in Section 2.2, the scheduler is executed on each processor simultaneously. Thus load balancing is invoked on each CPU independently and we will refer to the processor that is currently executing the code as the *local* one, whereas all other CPUs are referred to as being *remote* ones.

Whenever load balancing is invoked on one of our cores, we iterate over all remote processors in the system, fetching their appropriate frequency value and the load of the tasks currently running on this particular CPU. As we are interested mainly in the response times of tasks, we calculate the possible future ratio between processing power and timeslice lengths of the local node and the actual ratio of each remote node. The possible future ratio is composed of the processing power of the corresponding processor, the timeslices of all current active tasks in the runqueue of this processor, and the timeslice of the task that would be the one to be migrated from the remote CPU to the local one. This is an important aspect as we are interested in the fact if we obtain better response times by migrating a particular task to our local CPU, compared to the current situation.

By comparing the two ratios to one another, we aim towards locating the core which achieves worse response times than our local one. This remote node is considered to be the node from which we will pull one or more tasks.

A possible task to processor assignment of six tasks on an eight-fold multiprocessor system where each processor runs at a different speed, ranging from 275 MHz up to 2 200 MHz is shown in Figure 3.1.

Note how the task aluadd2 is soon migrated from CPU-2—running with 825 MHz—to the fastest processor in the system, CPU-7, with 2 200 MHz. Although the latter is already loaded with another task (aluadd3), the response time for both tasks increases on average if they are assigned to the faster core instead of each having a processor for themselves. We further observe that once task aluadd4 exits the system, aluadd2 is pulled by the now idle CPU-6, as it may serve with better response times compared to CPU-7 which is loaded with two tasks at that point.

Again, we now face the situation that the fastest core is loaded with only one task and as soon as this processor notices the imbalance compared to CPU-3, it pulls the task memrw to itself. And once again, as soon as aluadd2 on CPU-6 exits the system, the scheduler detects an imbalance and assigns task aluadd3 from CPU-7 to CPU-6.

Although we measured the task to processor assignments every 100 ms, we plot only every fifth point of each sequence to improve readability. For the same reasons we connected the points with lines, even though the measurement is based on distinct values only.
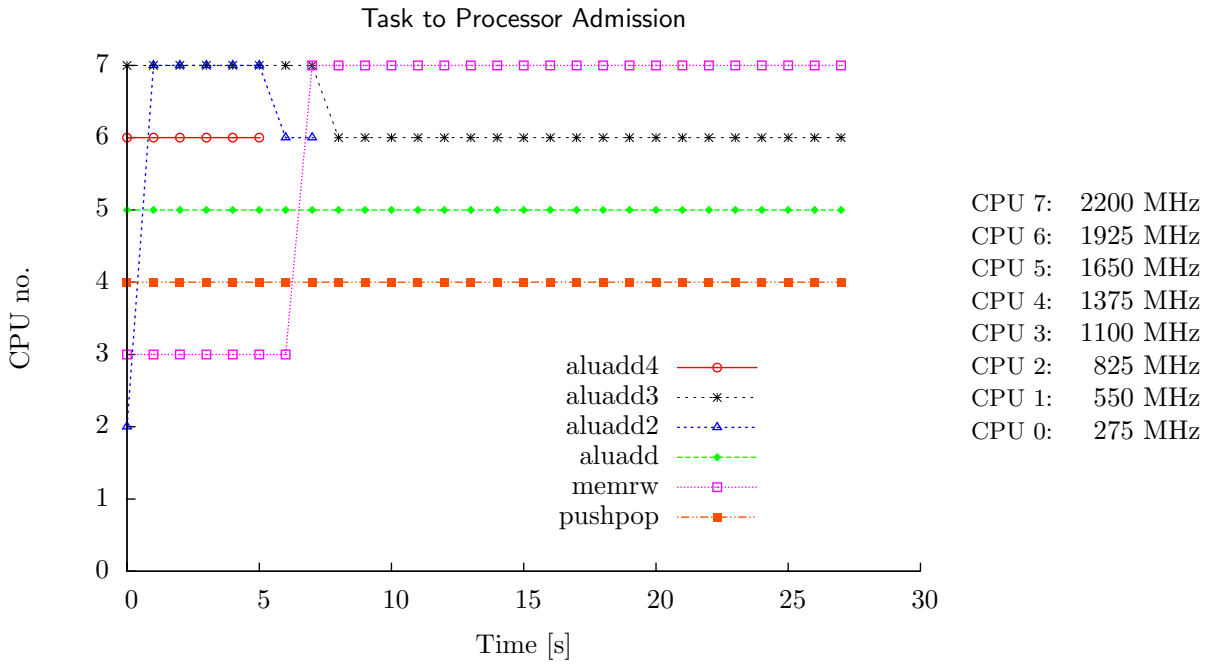
26

Figure 3.1: The first strategy which deals with best possible response times for all tasks on average applied to an eight-fold multiprocessor systems where each of the CPUs run at a different speed, ranging from 275 MHz up to 2.2 GHz.

3.2 Highest Throughput based Design

Another strategy is based on maximizing the system's throughput rate. We declare throughput in a multi-processor system to be the amount of data that can be computed in a certain amount of time. As a rule of thumb we can calculate the amount of instructions being executed on all CPUs within a certain time frame Δt.

This simplification doesn't deal with the specific runtime characteristics of individual tasks such as different levels of memory-intensity. Consider two distinct tasks, one very CPU bound by using very few memory access instructions, whereas another one deals frequently with instructions that need to operate on data from memory locations. Accessing memory is significantly slower than just working on CPU registers. Therefore, we note that memory-intensive applications spend much more time waiting for the data from memory and cannot execute as many instructions as CPU-intensive tasks, although both operate on equally fast processors. Nevertheless, this simplification serves well enough in order to get a basic understanding of the term throughput.

Throughput centers on the entire system rather than with individual processors or tasks. That is, no processor should be left idle before assigning a second task to any one processor by definition, no matter how big the differences between these two cores are.

To get a better understanding of the differences between optimizing for the best mean response time and the highest throughput rate, consider having two processors of very unequal speed, e.g. 1 000 MHz and 200 MHz, and two tasks with equal timeslices of

50 ms. Assume a processor executes 1 000 instructions per MHz and ms, such that the faster CPU executes 50 000 000 instructions of the above-mentioned task within each timeslice, whereas the slower CPU could only execute 10 000 000 instructions per timeslice. If both tasks were to be assigned to the faster CPU, both may be served with an average execution rate of about 25 000 000 instructions per timeslice as they would be preempted every other 50 ms by the other task. This would be quite beneficial for the task that would have been initially assigned to the slower core. Therefore, in terms of mean response times we would favour the latter setup. However, in terms of a maximized throughput rate we observe that assigning both tasks to different processors would result in a total of 120 000 000 instructions for two timeslices, compared to the 100 000 000 we would achieve by assigning both tasks to the same processor and leaving one core idle.

Therefore, none of our processors should be left idle before any other one is loaded with more than one task. However, when we face a workload with more runnable tasks than processors in the system, we need to find a suitable strategy to distribute these tasks among all processors.

In case all processors are allocated at least one task, the total system's throughput rate is already maximized because we cannot increase the amount of data to be computed by the system in a certain time period. However, we can consider maximizing the system's average response time under the condition that as many processors will be utilized as possible, that is we concentrate primarily on a maximized throughput rate and only afterwards on each task's individual response time.

In order to do so, we assign tasks to processors stepwise, going from the fastest CPU to the slowest, such that each processor serves only one single task. Once all CPUs are loaded with one task each, any further task is assigned to a processor that yields the best response time for this particular task.

Consider again a dual-core system with each core running at a different performance level. If we were to assign three tasks to those two cores we would certainly ensure each core to be loaded with at least one task, no matter how fast or slow a particular core is. With respect to the mean response time we should assign the third task to the faster core because even though both tasks would have to share its processing power, this core still serves with a better computational power for each task, which finally results in a greater amount of data that can be computed in a certain time period on average.

In case the throughput rate is maximized and all processors are loaded with one or more tasks each, we intend to utilize faster cores proportionally to their actual power, i.e. a processor running two times as fast as its slower counterpart should be loaded with two times as many tasks.

Therefore, we look at the ratio between any two processor speeds, and in order to obtain the *CPU ratio*, we divide the speed of the faster CPU by the speed of the slower one. In regard to the two CPUs and their tasks' response times, we then want to achieve a distribution of tasks among both CPUs that matches their speed ratio best. Before we provide a formula which serves as an axiomatic solution to this subproblem, we want to traverse an incremental solution step by step.

First of all, we would need to sum up all processes of both runqueues to a total number

of n. Starting from this total number, $n + 1$ possible combinations of task-to-processor assignments exist, namely:

$$n : 0 \qquad n - 1 : 1 \qquad n - 2 : 2 \qquad \ldots \qquad 2 : n - 2 \qquad 1 : n - 1 \qquad 0 : n$$

Based on these combinations we aim towards finding the one that matches our CPU ratio the best. This assignment is called the *task ratio* accordingly. However, it would be rather inefficient to test all of these ratios for the best possible setup. As we mentioned earlier, some should never occur such as idle processors in highly loaded systems or slower processors that serve more tasks then their faster counterparts.

There are basically three different objectives that should be considered in order to avoid unnecessary load comparison calculations:

- First of all, we do not consider cases in which the slower CPU has more processes running than the faster one, as this will never maximize the system's average response times.

- In case of a 1 : 1 or 1 : 0 setup as seen from the faster CPU, we are already balanced and do not need any further calculations.

- Apart from that, we do not consider the case of an $i : 0$ setup either, if $i > 1$.

Although this reduces the maximal possible calculations to nearly half of the original number stated above, we present a formula from which the values can be directly obtained. Within this formula, we use n to denote the total number of tasks on any two CPU's and x to denote the number of tasks that would have to be assigned to the faster core in order to accomplish the given CPU's ratio which we specify as y.

The number of tasks on the slower core can be directly derived by subtracting the number of tasks on the faster core from the total number of tasks. Having two processors in our system with corresponding frequency values f_1 and f_2, this leads to

$$\begin{aligned}
\frac{n - x}{x} &= \frac{f_1}{f_2} = y \\
n - x &= xy \\
n &= x(y + 1) \\
x &= \frac{n}{y + 1}
\end{aligned} \qquad (3.5)$$

Using equation 3.5, the scheduler can quickly derive the best possible distribution of tasks among two processors and hence decide if a current setup fulfills the needs of optimal mean response times by still guaranteeing to achieve the best possible throughput rate.

This strategy works well for comparison calculations between two different processors with two runqueues. To make it work well in a multi-processing environment with more

than two CPUs, we try to find the current worst setup between any two processors by locating the biggest divergence between the corresponding CPU and task ratios. We will stick to the same scheduling methodology as in the preceding section, namely one local processor and several remote ones. The local processor is again the one which currently executes the scheduler and which tries to detect any imbalances compared to any of the other (remote) processors.

All we need to do is to iterate over all remote CPUs in the system and compare the corresponding value of CPU power and the number of running tasks to the local CPU. In case the current task distribution is optimal for the purpose of our needs, there is no need to migrate any tasks. The first imbalance found will be considered the worst setup until another one is found with an even worse task distribution in regard to the corresponding power ratio.

As long as load balancing is executed frequently and in short time intervals independently on every single CPU this strategy pays off soon and resolves any imbalances quickly.

For the sake of clarity we provide an example that illustrates the applied strategy.

3.2.1 Example

CPU-1 runs at 800 MHz; CPU-2 runs at 600 MHz. This results in a CPU ratio of $\frac{4}{3} = 1,33$. The following listing shows a total number of tasks for both runqueues and all possible task distributions with their corresponding task ratio. The task ratio closest to the afforementioned CPU ratio is set in bold.

- 4 Tasks – 3:1 (3) or **2:2 (1)**

- 5 Tasks – 4:1 (4) or **3:2 (1,5)**

- 6 Tasks – 5:1 (5) or 4:2 (2) or **3:3 (1)**

- 7 Tasks – 6:1 (6) or 5:2 (2,5) or **4:3 (1,33)**

- 8 Tasks – 7:1 (7) or 6:2 (3) or **5:3 (1,66)** or **4:4 (1)**

We omitted cases of one, two, and three tasks. The corresponding distributions would be 1:0, 1:1, and 2:1, respectively.

The last example with eight tasks illustrates that there may also be two possible solutions for the best distributions among two processors. In this case it might be worth considering the current task's distribution and computing which new setup would cause the least migrations.

Figure 3.2 shows an example of the runtime behaviour of six different tasks in an eight-fold multiprocessor system with four processors running at a speed of 275 MHz and four processors running at a speed of 2 200 MHz. Compared to the first strategy which was primarily focused on an optimal average response time, we can note that no CPU has more than one task to serve at any given time. The two tasks on CPU-1

(aluadd2) and on CPU-3 (pushpop) would have been scheduled on one of the faster cores together with another task while the lower four processors would have stood idle. This strategy prohibits such behaviour.
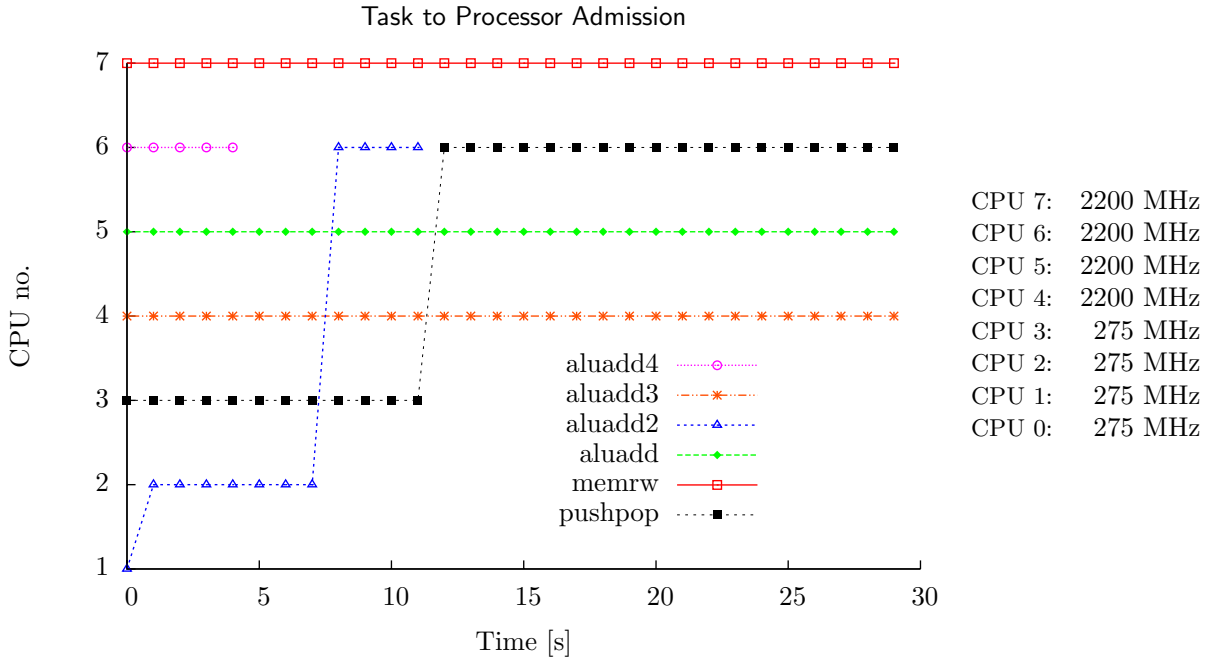

Figure 3.2: A strategy focused on achieving the highest possible throughput rate applied on an eight-fold multiprocessor system with four CPUs running at speed 275 MHz and four CPUs running at speed 2.2 GHz.

Instead tasks are assigned to processors based on the premise that as few CPUs should stay idle as possible. However, what we can derive from this measurement is that faster cores are preferred over the slower ones. We note how the task aluadd2 is migrated from CPU-2, which belongs to one of the slower four processors, to CPU-6, which is one of the faster four. And as soon as this task exits the system on CPU-6, a task named pushpop is migrated from a slower CPU as well.

3.3 Priority based Approach

The preceding sections proposed different solutions on how to assign tasks to processors in an asymmetric system, so that faster cores are utilized primarily in order to increase the total system performance. Neither of the designs took into account the actual task that was to be assigned to one of the CPUs and the user had little chance to interact with the system in order to prioritize one task over another in terms of processor assignments.

In this section we propose an assignment strategy that is based on process priorities. That is we assign tasks to processors based on their static priority, so that a high priority job runs on a faster core whereas a lower priority job runs on a slower core.

This approach lets users interact and exert influence on the system as tasks can be arranged with different priorities once they are started.

3.3.1 Assignment Problems

Whenever we search for a specific solution we need to agree on the problem domain and exactly what we want to solve. What is considered to be the best solution may vary greatly and some solutions to different problems are even mutually exclusive. For example finding an *optimal solution* differs, depending on whether we want to minimize the system's response time or whether we want to minimize the system's throughput, as we have seen in the last two sections.

Best minimal Response Time

The response time is meant to be the time from a stimulus until the user gets feedback from the system. In this regard it is better to schedule two tasks on one single processor in a dual-core system if one core is a great deal faster than the other one. In this case we are interested in the effective speed that each task obtains.

Assume the highest priority task in the system is always guaranteed to be the one to receive the most effective CPU power and that each remaining task receives as much effective CPU power as it deserves according to its priority compared to all other tasks in the system.

An approach based on these strict rules faces two major problems. Working in a dynamic environment where tasks enter and leave the system frequently, makes it difficult to find a suitable system-wide assignment and forces the scheduler to recalculate the best load distribution among all processors every once in a while. This may incur a sizable scheduling overhead.

Moreover we will notice numerous migrations over time, because in systems where the speed difference between two cores is small, entering or leaving tasks will oftentimes lead to necessary migrations which result in a ping-pong-like effect.

For example, consider a dual-core system with core one running at 800 MHz and core two running at 600 MHz. The first task is assigned to the first CPU. The second task with a lower priority will be assigned to the second CPU. In case a third task enters the system, we still want to guarantee the highest priority job the maximum CPU power. In order to achieve the best possible average response times, the first CPU needs to be shared by two processes, resulting in an effective CPU power of 400 MHz per task on this CPU and 600 MHz for one single task on the second CPU.

Regardless of the new task's priority, the scheduler is forced to perform migration. If the new task is the now highest priority job the scheduler assigns it on CPU two to provide it with 600 MHz but task two needs to be migrated from CPU two to the first CPU, as well. If the new task is not the new highest priority job amongst all three, it would be even worse, as the scheduler would have to migrate twice: task one from CPU one to CPU two and task two in the opposite direction.

By having a fourth task entering the system, the scheduler would again have to migrate all tasks around, because in order to achieve the best possible effective CPU power values, CPU two would be shared among two tasks, resulting in an effective CPU power of 300 MHz per process. That leads to the situation that the processor providing the best effective CPU speed changes again from CPU two to CPU one in this case and thus this is the CPU which is the designated one for the highest priority jobs. The highest priority job, however, was just migrated to CPU two in order to guarantee best performance values.

These three steps are illustrated in Figure 3.3. The length of a given process in the illustration indicates the resulting effective CPU power.
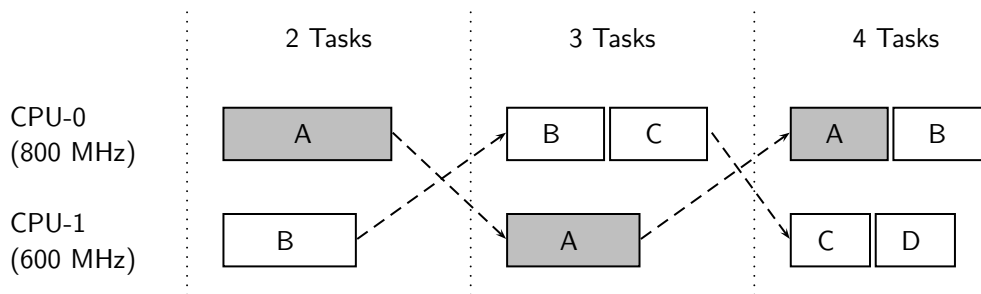

Figure 3.3: Priority-based scheduling depending on effective CPU power results in consecutive migrations whenever tasks enter or leave the system. The illustration above may differ slightly according to each task's priority.

This means that whenever only one task enters or leaves the system, we face numerous migrations which are necessary to fulfill the condition of best response times based on priorities.

Strict Priority based Best Effort

An optimal solution based on the latter approach relies on frequent migrations. Whenever necessary migrations are not performed, the resulting workload faces a kind of *priority inversion*, because a low priority job receives more effective CPU power than a higher priority one.

In order to avoid the aforementioned problems while still guaranteeing the highest priority task receives the best possible effective CPU power, we propose another approach that deals with two more generic aspects; first of all we always want to guarantee that we never face a priority inversion like the one mentioned above, i.e. the effective processor power must always be greater for a high priority job than the one provided for a lower priority job.

Besides this basic rule, we define a convention that requires that a faster core will never handle more tasks than any slower core in the system. It may handle an equivalent amount of tasks, but never more. Given this convention, a faster core will always serve high priority processes exclusively as shown in Figure 3.4.
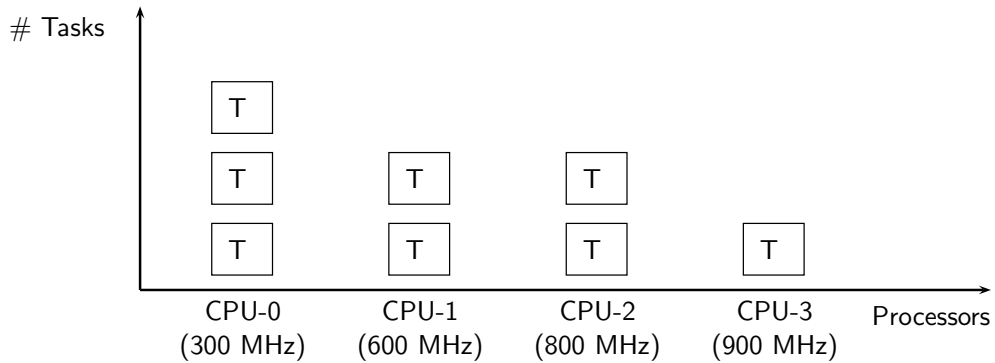
Figure 3.4: A best effort mechanism based on priorities would always guarantee a high priority job to be assigned to a faster processor than any lower priority job in the system. Furthermore no faster core would ever have to serve more tasks than any of the slower cores. The distance of assigned tasks between any two neighboring cores should never exceed one.

However, this is only a best effort approach since we do not compare the actual processor speeds to one another but only take into consideration whether one processor is faster than the other. This may lead to suboptimal scheduling decisions regarding response times as we do no distribute tasks among processors such that effective processor capacities are not completely utilized.

3.3.2 Principal Operation

Building a scheduling mechanism for heterogeneous systems based on priorities should ensure that a high priority task will always be assigned to the faster processor compared to any lower priority task, thus avoiding a priority inversion. Besides this basic demand, we want to guarantee reasonable response times. The last section showed which assignment problems we face if we either work towards a guaranteed best minimal response time at any given point in time, or a simplistic strict priority based best effort approach. This section proposes an approach that does not face any of the aforementioned problems, but guarantees the highest priority jobs are assigned to the fastest cores in the system while still yielding good average response times.

Dealing with response times leads again to the notion of effective processor power per task on each core. Section 3.3.1 showed that trying to obtain the maximum possible effective CPU power for each task based on priorities, leads to an unacceptable migration overhead in a highly dynamic system. Therefore we aim towards a mechanism that avoids such ping-pong effects but yields significantly better results than the aforementioned best effort approach in terms of response times per task.

We want to achieve a good balance of both objectives. To fulfill these needs we factor the ratio of the specific power values of two CPUs into the calculation. Dealing with integral numbers of tasks we only consider the *floor* of each ratio, i.e. given two CPUs with speeds of 800 MHz and 600 MHz, we end up with a ratio of 1, whereas two CPUs

with speeds of 1 000 MHz and 300 MHz result in a ratio of 3.

A given ratio of n, states that we may distribute n times more tasks on the faster core than on the slower one. The conservative floor estimation protects us from an unnecessary ping-pong migration overhead and still yields good average response times.

However, it is a strict convention that we do not compute the ratios of any two processors to one another. Rather it is necessary to compare only the power values of two *neighbouring* CPUs, that is two CPUs which are adjacent to each other in terms of speed.

Figure 3.5 illustrates the load balancing behaviour. A higher task number reflects a greater priority. Note that the initial task placement of tasks four and five was randomized. The important aspect is the fact that CPU-1 observes the imbalance between CPU-2 and itself and pulls the lowest priority task from CPU-2. Later, on the other hand, CPU-2 observes an imbalance between itself and CPU-1 as the power ratio between both CPUs is only 1:1 but the current task ratio is 3:1. Therefore CPU-2 pulls the highest priority task from its weaker neighbour.

We evaluated our design on an eight-way multiprocessor system where each of the CPUs runs at a different speed, ranging from 275 MHz to 2.2 GHz. Four tasks were started, each with a different priority. The results are shown in Figure 3.6. aluadd was the task with the highest priority and was scheduled on the fastest processor CPU-7, whereas aluadd2 was scheduled on the second fastest processor, CPU-6, as its priority was the second highest in the system. pushpop was the task with the lowest priority and hence scheduled on the CPU with the highest possible speed which was not already occupied by another task of higher priority.

We note that aluadd2 exits the system after approximately 5 seconds and memrw, the task with the next highest priority switches to CPU-6 immediately. As soon as pushpop on CPU-4 notices that a faster CPU is idle, it switches to CPU-5.
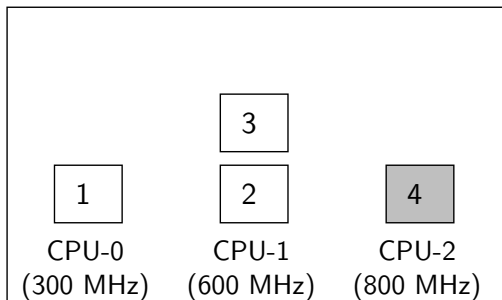
(1) Three tasks are distributed evenly among three processors.
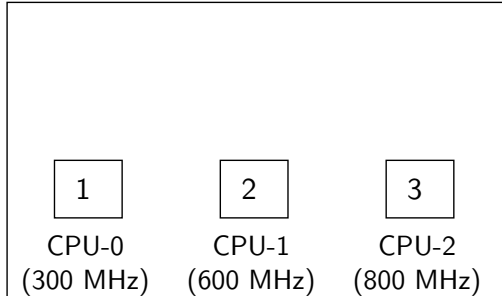
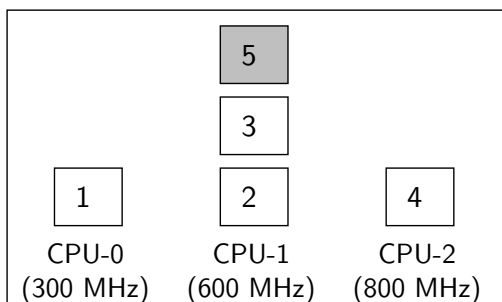**(2)** The new highest priority task 4 enters the system on CPU-2 which results in an imbalance between CPU-1 and CPU-2.
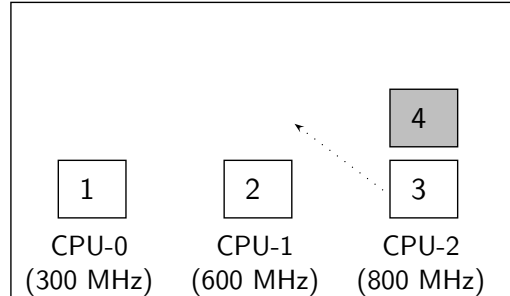
(3) Task 3 was pulled by CPU-1 hence load balancing is complete.

(4) A fifth task enters the system with priority 5.

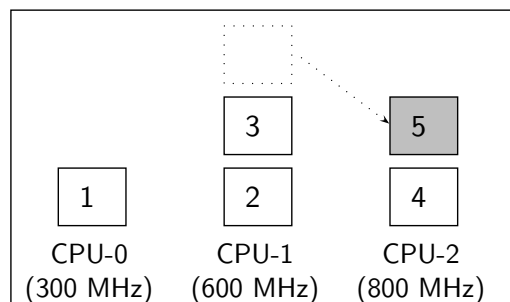**(3)** One task is migrated from CPU-1 to CPU-2.

Figure 3.5: An illustrated example of priority based load balancing on a three-way multiprocessor system.
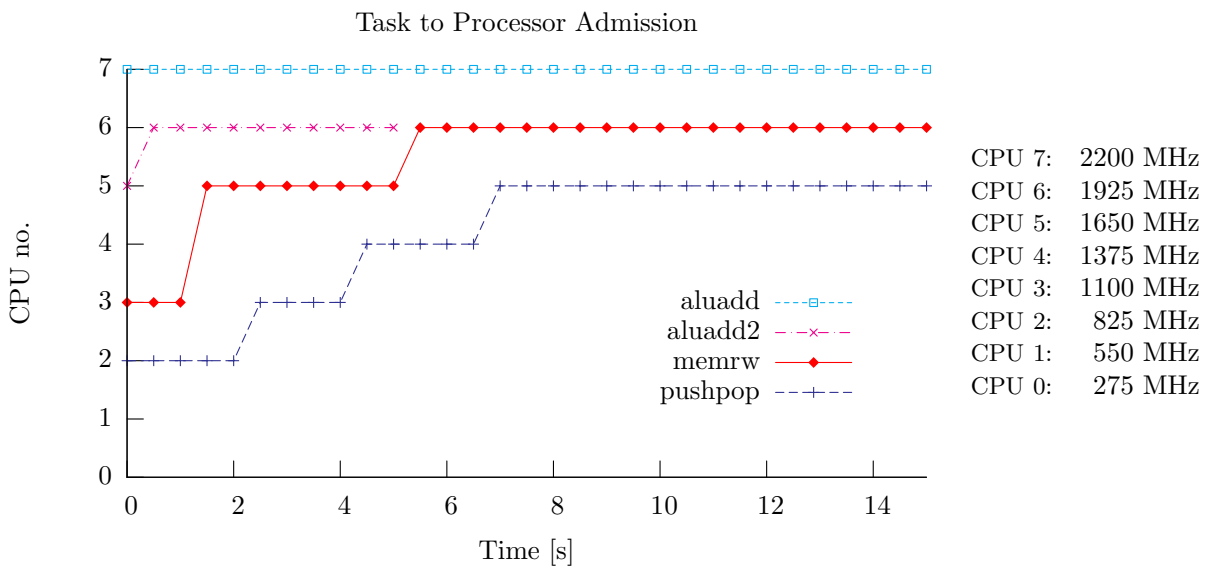
Figure 3.6: Task to processor assignments achieved by an approach based on the task's priorities and applied on an eight-fold multiprocessor system, each CPU running at a different speed.

4 Implementation

We implemented the strategies described in the preceding chapter—best throughput rates, best response times, and a priority based design—for the Linux kernel 2.6.21. Our goal was to keep most of the original scheduler code as it is, because we want to prove that today's SMP operating systems such as GNU/Linux can be easily adapted to fulfill the need to be asymmetry-aware and do not need to be entirely rewritten. All changes amount to roughly 600 lines of newly inserted code and about the same number of deleted lines of code. The additions are completely architecture independent, except for the part that actually picks the current CPU's frequency, which is only implemented for the following three architectures: *i386*, *ia64*, and *x86_64*.

4.1 Relevant Parts of the Linux Scheduler

Linux uses the round-robin scheduler algorithm for non real-time tasks, that is, every process is assigned a time quantum and on expiration of the time quantum, a new process is dispatched, hence processes are preemptible. The length of a given time quantum is directly related to the process's priority, such that high priority tasks get larger timeslices than low priority tasks. The priority itself is composed of a static part and a dynamic part. The static priority is inherited from a process by its parent and can be adjusted by a user via the `nice()` system call. The dynamic part is used in order to adjust the static priority according the process's runtime behaviour. That is, a process gets a bonus if it had to wait for a long time or is penalized if it has run on the CPU for a long time.

The Linux scheduler manages two sets of runnable processes for each CPU, a set of *active* and a set of *expired* ones. Even though both types of processes are ready to run, the ones in the expired array have already exhausted their time quantums and have to wait for all other processes in the active array to conclude. That means, that whenever a process is preempted by another process but didn't finish its timeslice, it remains in the active array, whereas processes that have exhausted their timeslices are transferred from the active to the expired array. This mechanism helps avoiding process starvation as low priority processes are eventually allowed to run, no matter how many high priority processes compete for the CPU. Once no more processes are present in the active array, the scheduler exchanges pointers pointing to the active and expired array.

Linux calls the expired and active queues *priority arrays* as they contain all tasks in arrays of linked lists. The size of each array reflects the number of linked lists and each list serves one of the possible 140 priorities in a Linux system, hence the name priority array. Figure 4.1 shows the section of a CPU's runqueue that deals with both priority arrays. Two pointers, `active` and `expired`, point to the corresponding priority arrays.

A priority array is composed of the following structure which contains three elements:

```
struct prio_array {
        unsigned int nr_active;
        DECLARE_BITMAP(bitmap, MAX_PRIO+1);
        struct list_head queue[MAX_PRIO];
};
```

First, the number of runnable processes in this array; then a 140 bit wide bitmap[1]; and at last an array of 140 linked lists. The bitmap represents the existing priorities of the runnable processes within the array, i. e. a set bit indicates a present process of this particular priority. Using architecture dependant assembly instructions, the scheduler uses `sched_find_first_bit()` to find the lowest order bit in $O(1)$. The returned index serves as a pointer increment to a list of processes with this particular priority.
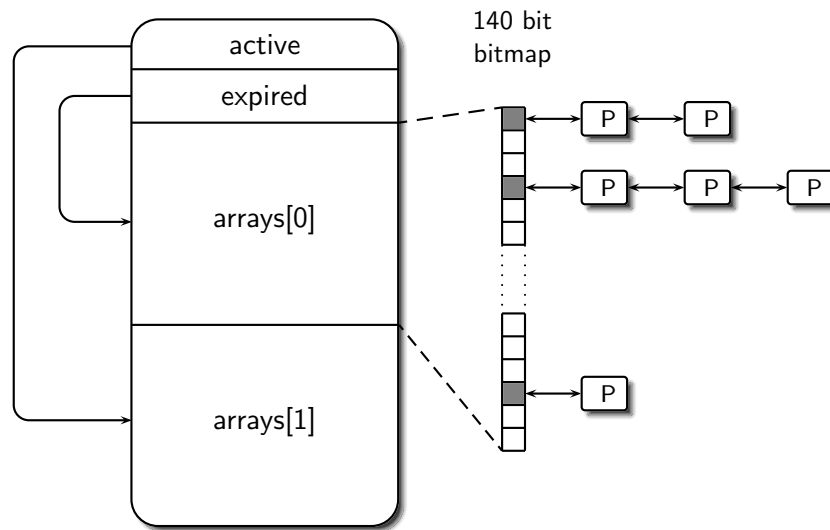

Figure 4.1: The runqueue's priority arrays section. The 140 bit wide bitmap indicates whether a task of this particular priority is present with a set bit. In this case all tasks with this priority can be accessed via a doubly linked list.

The Linux kernel 2.6 shows major improvements in terms of scalability in multiprocessor environments compared to its 2.4 predecessor. These performance gains were achieved amongst other things by the allocation of one runqueue per CPU in multiprocessor systems[2]. The runqueue is the most important data structure of the Linux 2.6 scheduler. Besides other things it keeps track of the number of runnable processes on the CPU, as well as the CPU load caused by the present processes and pointers to the currently running task on the CPU (`struct task_struct *curr`); the idle thread; the aforementioned priority arrays; the scheduling domain in which this CPU is located (please see Section 2.3 for an illustration) and the migration thread.

---

[1]here one extra bit serves as a delimiter.

[2]The 2.4 scheduler used one global runqeue for all processors.

The main concern of the SMP-aware Linux scheduler is an even distribution of tasks among all available processors and any avoidance of biasing towards one CPU. The decision which task is to be executed on which CPU is affected by the underlying type of multiprocessing system, the current load of the available CPUs, and the load weight of the task.

4.1.1 Load Balancing for Multiprocessor Systems

The Linux scheduler is responsible for an even distribution of tasks among processors. To fulfill this requirement there are two points in time when the scheduler reviews current configurations. The first one occurs with an initial task placement, that is whenever a new task is created and started. This is done primarily by `sched_fork()` and `sched_exec()`, each of which calls `sched_balance_self()` with a special flag indicating whether the process was forked or if a binary image is to be executed. The second one handles task migration among processors during runtime in a multiprocessor environment. A `rebalance_tick()` function is invoked by `scheduler_tick()` once every tick and iterates over all processors in a given scheduling domain to determine whether the `load_balance()` function should be invoked or not.

The basic idea of the load balancing design is based on a so-called *pull strategy*. Once an imbalance is detected, a CPU does not actively push tasks to remote CPUs. Rather, a CPU iterates over all groups of its scheduling domain and whenever it notes that the remote group is loaded with more tasks per CPU than itself, it initiates a task migration towards itself. The three basic functions that implement this mechanism are `find_busiest_group()` to retrieve the group that has the highest load within the scheduling domain; `find_busiest_queue()` to search for the busiest queue within that group; and finally, `move_tasks()` to pull tasks from the CPU belonging to the busiest queue towards the CPU currently performing load balancing.

We observed a frequent intercommunication regarding load comparisons between any two processors in the system which leads to an exchange of information at least every two to three seconds, depending on the arrangement of cores.

4.2 Asymmetry-aware Scheduling

We extended the Linux scheduler to be asymmetry-aware in the sense that it knows about a CPU's frequency at every point in time and takes the speeds into consideration when making scheduling and assignment decisions.

Therefore, the very first implementation detail had to deal with a proper extension of the current scheduler's runqueue data structure to reflect the actual CPU's frequency. To obtain the frequency values at runtime, we used an already present kernel function called `cpufreq_quick_get()` which we slightly modified so that it could be used at each load balancing invocation. As mentioned earlier in this chapter, this function is currently only available for the three architectures *i386*, *ia64*, and *x86_64*.

The function is called frequently during load balancing to keep track of any recent changes in the underlying system. The frequency value obtained is stored in the CPU specific runqueue data structure, from where it can easily be accessed for any further calculations.

4.2.1 Passive Load Balancing

Whenever load balancing calculations are invoked by the scheduler, the group with the highest load at that point in time is searched for, in response to the aforementioned pull strategy. The scheduler executes the function `find_busiest_group()` which is the major entry point for our adaptations.

The goal of all implemented strategies was to achieve a self-organizing system that performs the necessary load balancing steps in a suitable amount of time. Unlike the native scheduler implementation, we do not aim to find the least loaded group in terms of the number of running tasks on any given core, but rather to compare processor speeds with number of running tasks and the priorities of tasks.

In the original implementation, Linux simply uses a constant (`SCHED_LOAD_SCALE`) as a load balancing factor per processor which summed up corresponds to a group's specific `power` value. Using this constant the scheduler compares groups of different sizes and their current load with tasks and decides based upon this information where tasks, if any, need to be migrated to. However this implementation does not take the actual power values into account. We changed that behaviour in the way that the `power` value reflects the current CPU's frequency value. If a group is composed of more than one CPU, we adopted the already present design which simply sums up all CPU's power values of this group. These changes were applied to all three asymmetry-aware design proposals.

The load balancing mechanism within `find_busiest_group()` is divided into two major parts. At first the scheduler iterates over all present cores in the scheduling domain. All necessary values for load comparisons are obtained herein, such as the number of tasks in the runqueue, the CPU power, and the load of all tasks in the runqueue. We mentioned earlier in this paper that each CPU in a multiprocessor system runs its own instance of the scheduler, such that all instances are executed simultaneously. This means that any such instance considers the runqueue belonging to its processor to be the local one, whereas all other runqueues are considered remote ones.

In order to find optimal solutions for load comparison calculations in all of our proposed designs, a new data structure was implemented which yields a global view of all necessary remote CPU variables. The data structure is designed to be used as a linked list, such that the scheduler can iterate over all remote CPUs and fetch CPU-specific data without the necessity of having to access each remote runqueue every once in a while.

The data structure is defined as

```
struct asym_list {
        int cpu;
```

```
        unsigned long load;
        unsigned long pwr;
        unsigned long nr_running;
        struct sched_group *group;
        unsigned long best_prio;
        unsigned long worst_prio;
        struct list_head list;
    };
```

Every remote CPU within our domain is added in sorted order to the list with the CPU's power as the sort key.

The second part of this function is the actual load balancing calculation. Based on the different objectives of each of our three proposed designs, the code slightly varies. The scheduler should ultimately obtain the pointer to the remote group that is considered to be the busiest one, and from which it can migrate tasks towards itself. In case the local CPU is not loaded less than any of the remote CPUs, the scheduler discards this calculation and signals the load balancing function that no migration has to take place.

The designs based on the approach to achieve the highest average response times and the one to achieve a maximum throughput rate used the number of active tasks in each runqueue, the CPU power values, and the current load of each CPU in order to start load balancing calculations as described in Section 3.1.3 and 3.2.

The priority based strategy was designed such that a slower CPU pulls the highest priority task of a faster CPU, if any of the local tasks is of a higher priority than any of the remote ones. A faster CPU in contrast always searches for the highest priority process of any slower CPU. We added functionality to retrieve the highest and the lowest priority of any given runqueue in order to fulfill these conditions.

4.2.2 Active Load Balancing

It turned out to be insufficient to rely on passive load balancing where a highly loaded CPU has to wait for a lightly loaded CPU to help it out and migrate tasks towards it. The first reason for this is that sometimes tasks are not migrated fast enough. As we mentioned earlier in this chapter, CPUs share their runqueue characteristics frequently, but some combinations do so only once every two seconds. Even then it is not guaranteed that all environmental circumstances hold for a successful migration. Therefore, especially short running processes will seldom profit from the design.

Finally, we observed major problems when using the passive pull strategy while dealing with a priority based design. As noted in Section 3.3 the slower core is assigned to pull the lowest priority job of a remote CPU's runqueue, once it detects an imbalance between the cores. The faster core must migrate the highest priority process towards itself from a slower core.

The original Linux scheduler implementation is designed to always pull the lowest priority process from a remote runqueue. Using passive load balancing in order to migrate the highest priority process of a given runqueue proved to fail too often. The reason for this anomaly is that high priority processes receive greater timeslices than

the lower priority processes. Thus, the high priority process is much more likely to be the one that is currently running on any given CPU. The Linux scheduler tries to avoid moving an actively running, so-called hot, task from a CPU. It does so by querying the function `task_running()` if a task is currently active or not, and in case it is, does not take this task into consideration for migration.

We extended the Linux scheduler to use active migration by invoking the `move_tasks()` function with a special flag, indicating whether the highest priority process is to be migrated or not, and with the number of the local CPU to which the remote CPU should push tasks. Once these two fields are filled, we wake up the migration thread of the busiest CPU which will immediately notice the flags set for active load balancing and migrate the task towards the runqueue of the former local CPU. As active migration happens immediately and can be used to migrate the highest priority process of a runqueue, it yields better results even for short running tasks.

5 Experimental Results

We evaluated our different implementations of asymmetry-aware schedulers to show how the performance of only SMP aware systems are negatively affected when running on a heterogeneous system. We provide results that prove that an asymmetry-aware scheduler is necessary in order to benefit from the system's hardware.

5.1 Test Setup

Our test environment was based on an 8-way Pentium 4 Xeon multiprocessor consisting of two NUMA-nodes containing 4 processors each. Every single processor supplied 2.2 GHz but could be throttled to one of eight possible speeds ranging from 275 MHz to 2.2 GHz. To achieve a real asymmetric setup, we used the clock modulation mechanism as described in Section 2.1.2. Although Hyperthreading was available on all processors, we disabled it completely in order to focus just on the case of different processor speeds.

To evaluate our implementations we used different benchmarking programs, ranging from simple CPU-intensive programs to much more sophisticated applications from the SPEC CPU2006 suite. Table 5.1 on the following page summarizes the programs along with a short description of what they are meant to compute.

In order to avoid any influences of the measuring tasks on our actual test suite, we ran all these measurement tasks on CPU-0 permanently. This can be achieved either via a Linux system call, namely `sched_setaffinity()` or via an open source utility called `taskset`. This was an important precondition as some of our tools monitored the benchmarking programs permanently and within short time intervals, such as that they would have been considered as running tasks by our scheduler and would have accidentally prevented some benchmarking tasks from being migrated to a particular CPU.

5.1.1 Performing Measurements

Linux provides several interfaces to export information to user space or to let users interact with the kernel directly. Some of those interfaces are implemented by using a virtual filesystem such as `/proc` or SysFS under `/sys`. The main advantage of these filesystems is that they do not occupy permanent storage on disk, but rather are an in-memory filesystem, whose entries are created only after a corresponding `read` or `write` call was issued by the user. This makes it then highly dynamic but avoids a constant runtime overhead that would have been caused if the kernel was to update the filesystem entries frequently.

Program	Language	Description
memrw	C	Memory Reads/Writes
aluadd	x86 assembly	Integer Additions
pushpop	x86 assembly	Stack push/pop Operations
400.perlbench	C	PERL programming language
401.bzip2	C	Compression
429.mcf	C	Combinatorial Optimization
456.hmmer	C	Search Gene Sequence
435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
436.cactusADM	C/Fortran	Physics / General Relativity
459.GemsFDTD	Fortran	Computational Electromagnetics

Table 5.1: All programs used for the tests and evaluations. Program names beginning with a number belong to the SPEC CPU2006 suite.

The Linux scheduler offers runtime statistics via the `/proc/schedstat` file such as the number of times `schedule()` was called, the number of times `load_balance()` was called under different circumstances, or further individual statistics for all runqueues in the system. This interface can easily be adopted and extended. Besides the schedstats, the `/proc/<pid>/stat` file has proven to be quite useful. It contains the current state of the task with PID `<pid>` (S for sleeping, D for uninterrupted sleep, R for running, Z for zombied, or T for stopped or traced), the dynamic priority of a process, or the CPU the process is currently assigned to. Both files are used to perform benchmark tests and are periodically (e. g. every 100 ms) traced from a user space program.

5.2 Stability and Predictability

One of our objectives was to demonstrate that an operating system that is unaware of the underlying heterogeneous architecture produces results which are less stable and show a decreased predictability. The terms stability and predictability stand for the balance we can reach on average by multiple, possibly thousands or millions of executions of a given workload.

These attributes are very important for a number of different applications and workloads. For example, business applications or soft real-time applications profit from relatively stable and predictable runtime behaviour. Assume stock exchange software would behave unpredictably to a great extent—no broker would ever use it if he could avoid it.

Even home users are annoyed quickly if the runtime behaviour of frequently used tasks differ substantially in terms of occurrence and durability. For example, using a system that may be good enough on one day to watch a video but not on another one would be unacceptable.

Therefore, our applied strategies had to achieve a stable and predictable behaviour. We carried out the performance benchmarks multiple times such that all of them yielded results that do not differ too much from one another.

We ran tests with some benchmarking tasks of the SPEC CPU2006 suite in order to quantify the performance of our implementation against one that is currently not asymmetry-aware. Figure 5.1 shows an example of different benchmarking results. All tasks were executed five times on our 8-way multiprocessor system which was set up to use eight different speeds, ranging from 275 MHz for the slowest processor to 2.2 GHz for the fastest.
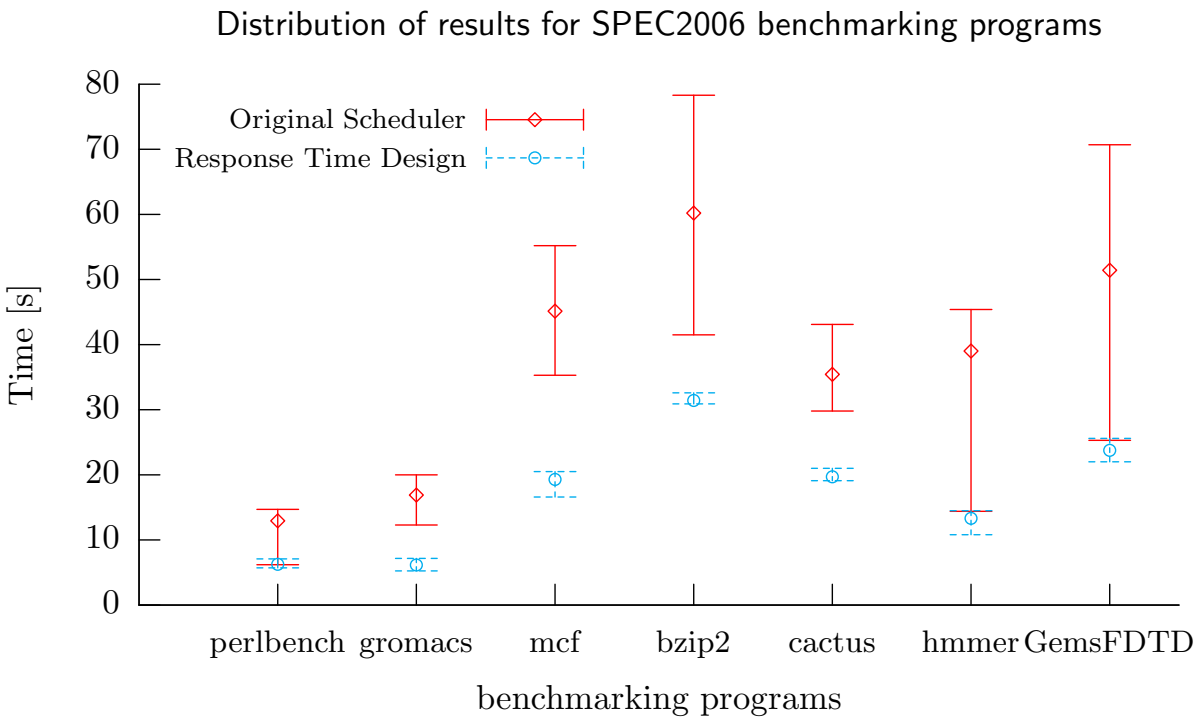

Figure 5.1: Different SPEC CPU2006 benchmarking programs tested with the unmodified Linux 2.6.21 scheduler and our best mean response times based design. Error bars indicate the varying degree of results for each implementation.

The figure illustrates how firstly we observe a performance boost by using the implementation that is based on achieving the best possible average response times. All results belonging to this implementation outperform the ones obtained by using an unmodified Linux 2.6.21 kernel. In terms of performance we need to especially consider the average values within each error bar.

However, by considering the predictability and stability of workloads, the figure pro-

vides even more promising results. The figure illustrates the maximum and minimum achieved values of any given benchmark with the average values located in between. We used error bars to show the diversity of results achieved by the original scheduler and the one that we adapted to be asymmetry-aware.

We note the distance between any two turning points for each of our benchmarking program and how it differs for both implementations, stating that our implementation is remarkably stable compared to the unmodified one.

5.3 Performance Gains

We showed that the original asymmetry-unaware scheduler fails in providing constantly good performance. We saw that the results obtained differ greatly but it is important to verify that the user profits from our new scheduler designs significantly.

Single task benchmarks are a good indicator in terms of performance and stability, but by using multiprocessor systems it is likewise more important to test the implementations against workloads that consist of multiple tasks. We used all three implementations that we proposed in this thesis—namely a scheduler based on a design which aims towards attaining best average response times, one that tries to achieve the highest throughput, and one that utilizes a task-to-processor-assignment based on each task's priority—to test them against the original not asymmetry-aware Linux 2.6.21 scheduler, by using different workloads.

The workloads are composed of one single program which is very CPU-intensive and is started in one or more instances. The program was built so that it needs 4.5 seconds to finish its calculations on the fastest core in the system and 32 seconds to be completed by being executed on the slowest core. Table 5.2 shows some of the results for our benchmarking program.

Strategy	1 instance			2 instances			4 instances		
	Total	Avg	Gain	Total	Avg	Gain	Total	Avg	Gain
Unmodified	19.34	13.85	—	19.36	12.74	—	19.36	11.09	—
Response times	7.12	6.59	52.4%	7.87	6.15	51.7%	8.80	6.14	44.7%
Throughput	12.23	7.21	47.9%	10.38	7.49	41.2%	12.26	7.52	32.2%
Priority based	12.26	8.40	39.3%	17.09	8.65	32.1%	13.98	7.72	30.4%

We conducted five measurements per workload and implementation. The workloads consisted of 12 different setups, ranging from one single task being executed, to 12 instances of one task being executed concurrently. For the purpose of clarity we picked only six workloads out of all 12 for Table 5.2.

The columns headed Total represent the total runtime of the entire workload, that is from the time when all instances of the task were started to the time when the last one

Strategy	8 instances			10 instances			12 instances		
	Total	Avg	Gain	Total	Avg	Gain	Total	Avg	Gain
Unmodified	35.64	12.18	—	33.50	10.95	—	35.16	10.31	—
Response times	10.26	6.68	45.1%	12.21	6.74	38.4%	13.72	6.74	34.6%
Throughput	16.78	8.70	28.6%	15.40	8.05	26.4%	15.60	8.09	21.5%
Priority based	17.31	8.87	27.2%	15.71	8.26	24.6%	23.53	8.70	15.6%

Table 5.2: Benchmarking results of one program run with a different amount of instances and with each particular scheduler implementation.

was completed. The Avg time represents the time that each instance was executed on our system on average. We obtained this value from the total accumulated time that the system spent executing all instances.

In order to see the system's improvement performance-wise, we compared the results of each workload's average values of the new designs to the one's of the original scheduler. The obtained difference is expressed as a percentage within each Gain column.

We observe performance boosts for each of our implementations, each of them being remarkably high in the range between 15.6% and 52.4%. From the numbers given above we derive that the design based on the best possible average response time will lead to the best results in terms of performance. This is what we expected when we elaborated the purpose of this design in section 3.1.

Besides that we further note a decreasing performance gain as the number of tasks increases. This is due to the fact that the original asymmetry unaware scheduler already distributes tasks to processors evenly. Although it does not deal with the actual frequencies of the processors, it will assign the tasks to CPUs in a comparable manner to our new designs. That is if there are more tasks than CPUs in the system, it becomes more and more likely that all cores are occupied by at least one task. This leads to a load distribution which differs only slightly from one processor to another.

Figure 5.2 illustrates the average results for all three scheduler implementations compared to the original scheduler. We note the instability of the original scheduler that is not asymmetry-aware, but which achieves two times better results on average than two of our proposed designs, namely when running three instances of our benchmarking program and when running six instances of it. However, in both cases performance gains of the original scheduler are rather small, especially compared to the performance loss that can be observed in all other cases.

We further note that our approach based on the best average response times is very stable, ranging from 5.74 seconds to 6.77 seconds, whereas the original scheduler fluctuates between 8.56 seconds and 13.86 seconds.

Figure 5.3 shows the maximum execution times for this benchmarking program for a
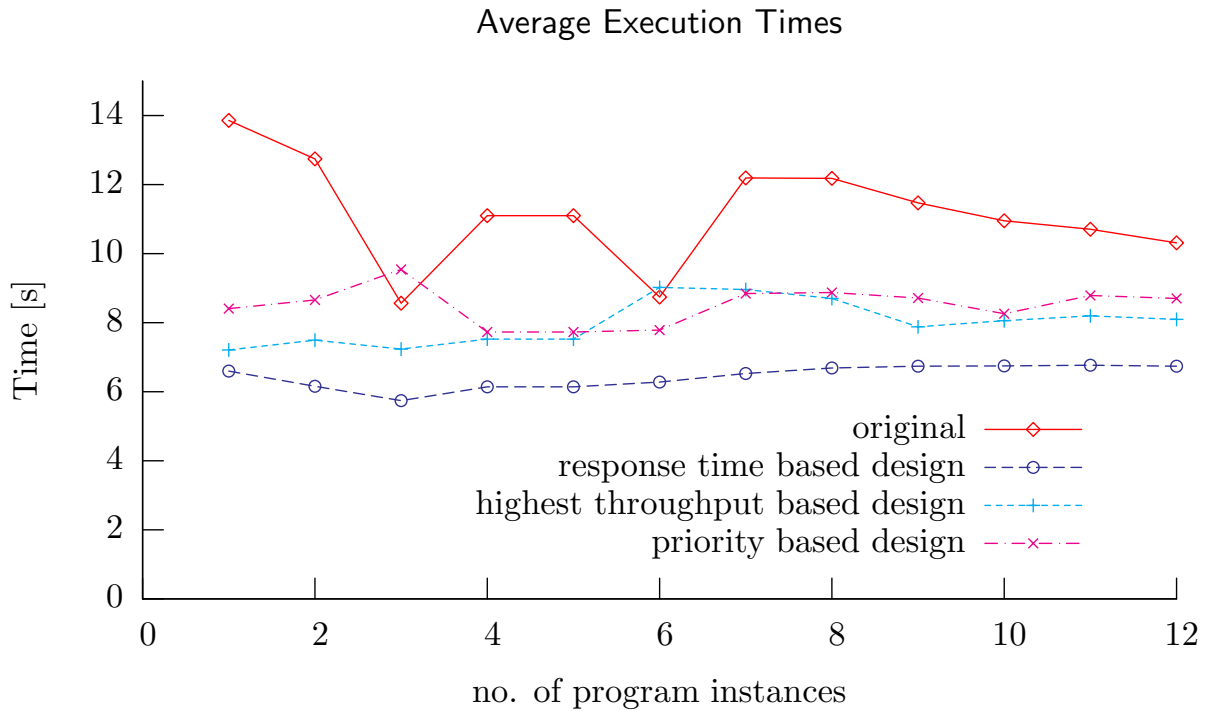
Figure 5.2: Benchmarking results of all three scheduler implementations compared to the original. The graph shows the *average* execution times for one program run with a different amount of instances on an eight-way multiprocessor system with eight different speeds.

different number of instances and all four scheduler implementations. From this figure we note the big discrepancy between the original scheduler and all asymmetry-aware implementations, especially when five or more instances are executed simultaneously on our eight-way multiprocessor system.

The original scheduler assigns tasks to processors based on the load of each particular processor in the system. In case multiple CPUs are idle with empty runqueues, the actual assignment of a particular task can not be foreseen and can be considered as being random. Therefore it is likely to see those tasks being assigned to processors ranging between the slowest and the fastest core which results in medium execution times.
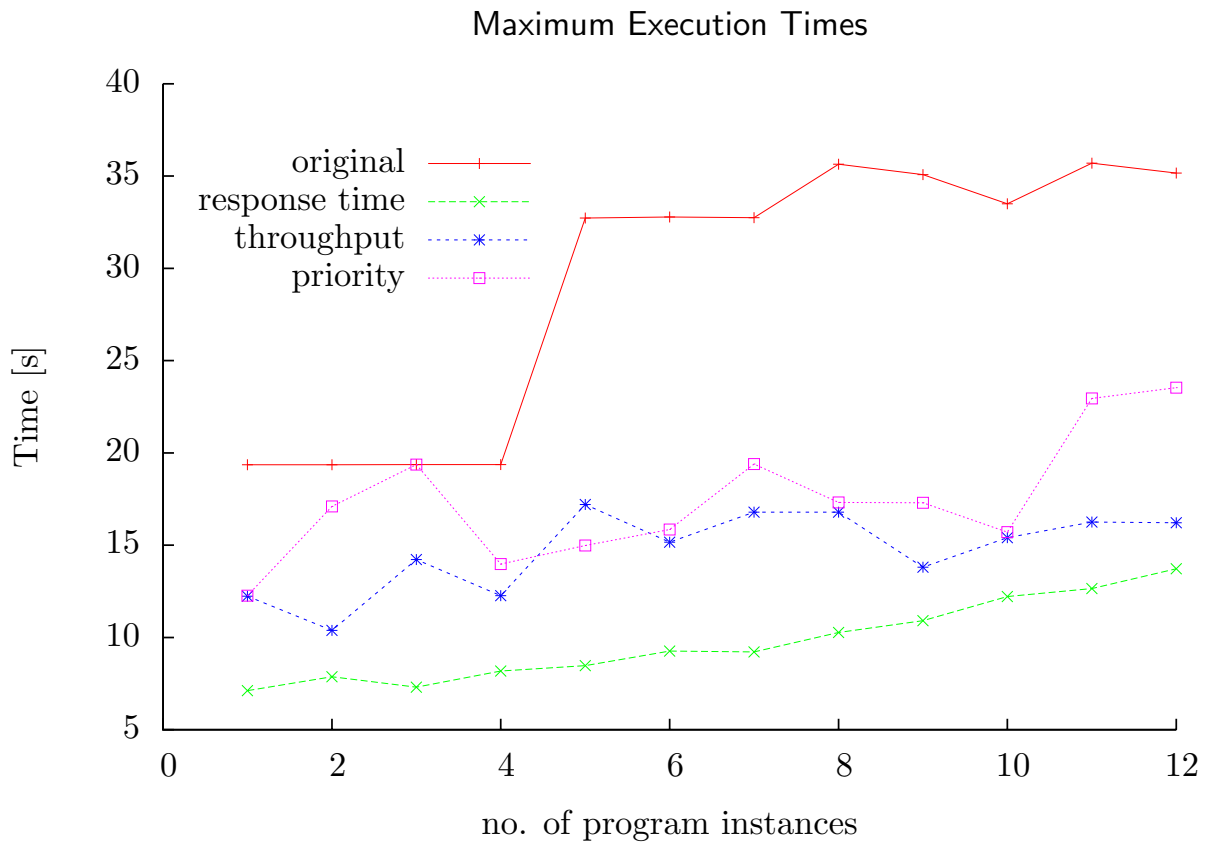
Figure 5.3: Benchmarking results of all three scheduler implementations compared to the original. The graph shows the *maximum* execution times for one program run with a different amount of instances on an eight-way multiprocessor system with eight different speeds.

6 Conclusion

6.1 Achievements

This thesis discussed the impacts of asymmetric processor speeds on SMP operating systems. We showed that an unaware operating system may suffer performance degradation and behave exceedingly unpredictably. We evaluated different designs that make an operating system aware of the underlying heterogeneous processors. The scheduler is best suited in order to benefit from the asymmetric setup.

Based on different objectives we proposed three different scheduler implementations which can be used to either guarantee best average minimum response times for all tasks in the system, or to achieve the highest throughput rate for the entire system, or by being priority bound to an extremely high degree.

Using an asymmetry-aware scheduler we obtained

- A greatly increased system performance for a variety of different workloads

- A much higher degree of stability and predictability of each individual workload in terms of performance and processor utilization.

All proposals enable the operating system to profit from the asymmetry in terms of performance to its full potential. Tasks will be assigned to faster processors preferentially and load balancing strategies ensure a balance of tasks among processors strictly according to the objectives of the specific design. These assignments and load balancing strategies further avoid a more random distribution of tasks which would lead to unpredictable performance results.

6.2 Summary

Modern computer systems are equipped more and more with an increasing number of CPUs. In the future we may observe multiprocessor systems that are composed of cores running at different speeds, hence asymmetric systems. Such systems exhibit multiple benefits, such as the potential from a business point of view to provide more cores per system at a lower price. Several simple low performance cores can be put on one chip together with some high performance cores. While the simpler cores are very cost efficient and provide good parallel performance, a few complex cores could be used to provide good serial performance.

Asymmetric systems are also interesting from an architectural point of view as they offer a lot of convenient options for system designers. Provided that all cores use the

same instruction set architecture, the architect may mix some new high performance cores together with some older ones. Low performance cores may serve with a better MHz per Watt ratio through a lower heat and power dissipation.

Our approach aims towards making today's SMP operating systems asymmetry aware. It considers primarily the system's process scheduler as the main entity to benefit from the underlying hardware instantaneously and continuously.

For this, we proposed three different possible scheduler adaptations that will utilize the system resources as best as possible. We determine either the best mean response times for all tasks in the system, according to the available processors, or the highest throughput rate for the entire system, or a strict priority-based approach which assigns the highest priority tasks to the fastest cores in the system, while still yielding good response times for all tasks.

All strategies were implemented for the Linux kernel 2.6.21. We conducted multiple tests on an eight-way Pentium 4 Xeon multiprocessor system where each of the CPUs could be throttled to one of eight possible performance levels.

Benchmarking results show that the operating system needs to be asymmetry-aware in order to avoid performance degradation and unpredictable behaviour. Depending on the particular strategy applied we observed performance gains of more than 50% and could prove that our designs are not subject to high fluctuations any more, but rather behave very stably and predictably in terms of system performance.

6.3 Future Directions

The currently evaluated designs are basically meant to be a proof of concept. Asymmetric systems are not currently under development by chip manufactures; therefore we face plenty of possible configurations that may be designed in the future.

The three scheduling strategies aim towards highly different objectives. While response times and throughput rates are treated equally in today's SMP systems, this is not the case for AMP systems any more. Faster load balancing mechanisms represent a limitation of our current designs. Migrations should take place immediately. This may be solved by changing the basic scheduling strategy to perform a more active load balancing by pushing tasks from one processor to another instead of relying on the pull strategy which sometimes leaves an imbalance unsolved for some seconds. Load balancing may also be invoked much more frequently within shorter time intervals. Our strategies are therefore much more applicable to long running tasks than for short running ones, although an implementation favouring short running tasks may be of more use for some interactive workload simulations.

We discarded some system specialities such as Hyperthreading abilities or the actual migration costs which differ highly between Hyperthreading nodes and NUMA nodes for example.

Future work may be directed towards a detection of highly parallelized applications in contrast to single-threaded applications. Heuristics can be designed in order to decide at which point in time a particular distribution of tasks has negative or positive effects

on the system's performance.

Each task's specific execution characteristic should be considered for scheduling decisions as well. Memory intensive or very I/O bound tasks may be served well enough on low performance cores, while CPU bound tasks should preferably be executed on the high performance processors. These mechanisms can be extended in order to achieve real power and energy savings.

Asymmetry-aware schedulers can be of particular interest for soft real-time applications and environments. Guaranteeing high priority tasks to be executed on the high performance cores could still enable the system to profit from the hardware if low priority tasks can use these cores during times in which no high priority task is active.

Bibliography

[BC06]   Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 29–40, New York, NY, USA, 2006. ACM Press.

[BR00]   Michael A. Bender and Michael O. Rabin. Scheduling Cilk multithreaded parallel programs on processors of different speeds. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 13–21, 2000.

[BRUL05]   Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, 2005.

[CBL+07]   John M. Calandrino, Dan Baumberger, Tong Li, Scott Hahn, and James H. Anderson. Soft Real-Time Scheduling on Performance Asymmetric Multicore Platforms. *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, 0:101–112, 2007.

[DM06]   James Donald and Margaret Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 78–88, Washington, DC, USA, 2006. IEEE Computer Society.

[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 workshop on ACM SIGOPS European workshop: beyond the PC*, page 9, New York, NY, USA, 2004. ACM Press.

[FSSN05]   Alexandra Fedorova, Margo I. Seltzer, Christopher Small, and Daniel Nussbaum. Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design. In *USENIX Annual Technical Conference, General Track* [USE05], pages 395–398.

[GG03]   Soraya Ghiasi and Dirk Grunwald. Aide de Camp: Asymmetric Dual Core Design for Power and Energy Reduction. Technical Report CU-CS-964-03, University of Colorado, Boulder, 2003.

[GKR05]   Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling for Heterogeneous Processors in Server Systems. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 199–210, New York, NY, USA, 2005. ACM Press.

[GRSW04]   Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of Both Latency and Throughput. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design (ICCD'04)*, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[KDG⁺04]   Ramakrishna Kotla, Anirudh Devgan, Soraya Ghiasi, Tom Keller, and Freeman Rawson. Characterizing the Impact of Different Memory-Intensity Levels. In *WWC-7: IEEE 7th Annual Workshop on Workload Characterization*, pages 3–10, Austin, Texas, USA, October 2004. IEEE Computer Society.

[KFJ⁺03]   Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[KTJ06]   Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[KTR⁺04]   Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[LY74]   Jane W. S. Liu and Ai-Tsung Yang. Optimal scheduling of independent tasks on heterogeneous computing systems. In *ACM 74: Proceedings of the 1974 annual conference*, pages 38–45, New York, NY, USA, 1974. ACM Press.

[MB06]   Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 403–414, New York, NY, USA, 2006. ACM Press.

[MBW05]   Andreas Merkel, Frank Bellosa, and Andreas Weissel. Event-Driven Thermal Management in SMP Systems. In *Second Workshop on Temperature-Aware Computer Systems (TACS'05)*, Madison, USA, June 2005.

[MWK04]   Tomer Morad, Uri Weiser, and Avnoam Kolody. ACCMP – Asymmetric Cluster Chip Multi-Processing. Technical report, CCIT, 2004.

[MWK+06]   Tomer Y. Morad, Uri C. Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE Comput. Archit. Lett.*, 5(1):4, 2006.

[NP02]   Jun Nakajima and Venkatesh Pallipadi. Enhancements for Hyper-Threading Technology in the Operating System: Seeking the Optimal Scheduling. In *WIESS* [USE02], pages 25–38.

[PMSD04]   Éric Piel, Philippe Marquet, J. Soula, and J.-L. Dekeyser. Load-Balancing for a Real-Time System Based on Asymmetric Multi-Processing. Internal Paper, April 2004.

[SPH+03]   Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 23(6):84–93, December 2003.

[SPM05]   Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Chip Multi Processing aware Linux Kernel Scheduler. In *Proceedings of the 2005 Ottawa Linux Symposium (OLS 2005), Volume 2*, pages 193–204, July 2005.

[USE02]   USENIX Association. *Proceedings of the Second Workshop on Industrial Experiences with Systems Software, WIESS 2002, December 8, 2002, Boston, MA, USA*. USENIX, 2002.

[USE05]   USENIX Association. *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005.