Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Masters Thesis

# Power Management in a Manycore Operating System

by
Dario Simone

Due date
25. August 2009

Advisor:
Akhilesh Singhania

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland

# Abstract

The search for managing the increasing power consumption of today's systems is still unbroken. The core idea of minimising power consumption while maximising the system's performance has been approached in earlier work using frequency and voltage scaling. However, with the coming of multi-core systems new constraints are imposed on power management solutions which make the single-core solutions difficult to use. A new challenge brought by multi-core, multiprocessor systems are the different sleep states for cores and processors.

In this work I present a new approach to power management for multi-core, multiprocessor systems. Using the different power levels of individual cores and whole processors of a multi-core, multiprocessor system, the system state is optimised in terms of power consumption and performance.

To show the possible gain for a system's power management I have implemented and evaluated a possible solution. To weight the cost of diminishing performance against a possible reduction in power consumption, a formal cost model of the system is created. The evaluation of the implemented solution shows a reduction in power consumption while keeping performance as high as possible.

# Contents

# Chapter 1

# Introduction

## 1.1 Problem Statement

Today's computer users and system administrators are increasingly power aware. Energy costs money and should be minimised. As the user base grows, the technology usually adapts. So, today's hardware usually supports many different power management features.

Processors in general support two different options to adjust power consumption and performance. The bulk of research has examined the use of frequency and voltage scaling and its impact on power consumption as well as performance. As will be shown in the literature survey (chapter 3) only most recent work in this field has approached the additional constrained given by the relatively new multi-core architectures. Conversely to single-core processors, multi-core systems come with constraints to the states of the individual cores. Therefore, frequency and voltage scaling research on single-core processors cannot be applied to multi-core systems using the same assumptions.

Apart from frequency and voltage scaling, modern processors support different sleep states. If combined in a multi-core, multiprocessor system, similar to frequency scaling new constraints apply to the processor sleep states. On such systems cores sharing a processor transition into a deeper sleep state (consuming less power) if all cores on the processor are sleeping. Thus, it is possible to reduce power consumption through intelligent placement of the threads on the different cores. Simultaneously, the same mechanism can be applied to control a system's performance by scheduling multiple threads on the same core. As will be shown in the literature review (chapter 3), this opportunity for power management has not yet been picked up in research.

## 1.2 Context

In this work I approach the problem of optimal thread placement in a multi-core, multiprocessor system running a non-real-time OS. I evaluate a running implementation on top of the Barrelfish [4] operating system. While the Barrelfish operating system structure is different from common operating systems, the approach and much of the implementation are designed to be applicable to

a variety of systems. My work imposes no additional restrictions on the system it is run on.

## 1.3 Contribution

In this thesis I present an approach on power management exploiting the possibilities given by intelligent placement of threads on cores. The solution considers the different levels of power consumption of a sleeping core depending on which other cores are sleeping. These power savings compared to the loss in performance if multiple threads are scheduled on the same core.

To this end I introduce a formal cost model to quantify the cost a system state has in terms of power and performance. Using that knowledge the system can then be put in a low-cost state. The cost model computes the possible power savings for different thread placements in view of the constraints imposed by a multi-core multiprocessor architecture. The model also weights the possible power savings of a system state against the system's performance in that state. Wheter more power should be saved or higher performance is needed can be set by a parameter to the model. Using this parameter the operating system or the user can adjust the systems power consumption and its overall performance.

The formal cost model is applied and optimised using the constraint logic framework ECLiPSe [9]. An implementation on the Barrelfish operating system using this model is provided and evaluated. In the best-case, the implemented solution reduces power consumption with a minimal impact on performance compared to a performance-optimal algorithm. Moreover, the impact of the model's parameter is shown both on power consumption and performance.

## 1.4 Overview

In the next chapter, I give background information on power management and the technologies used in my implementation. Chapter 3 summarises and discusses past work in the area of power management. Chapter 4 specifies the problem and presents the approach taken. Chapter 5 describes my implementation of the presented approach in the Barrelfish [4] manycore operating system. Chapter 6 evaluates the implementation using different schedules and settings and in chapter 7 I present my conclusions and possibilities for future work.

# Chapter 2

# Background

## 2.1 Power Management Overview

Power management comes in many different flavours. For some time every producer (be it software or hardware) tried to give his customers his own set of tools to reduce power output with a minimal impact on performance. Hardware manufacturers build devices which will spin down independently if the request rate is low. CPUs and chipsets for some time now implemented different interfaces to give operating systems or dedicated software the ability to control power saving and performance loss thereof. The Advanced Configuration and Power Interface Specification [1] has been introduced as a solution for a unified and coherent interface for the operating system to all power management aware devices and functionalities in a computer.

## 2.2 ACPI

ACPI (Advance Configuration and Power Interface) [1] was specified by different manufacturers to establish common interfaces for platform-independent configuration and power management. ACPI specifies solely the interface between hardware and software and the implied requirements of the two. The specification defines what has to be given and initialised by the hardware and what assumptions can be made on the software side. In the ACPI specification software is defined as an operating system component called *Operating System-directed Power Management (OSPM)*. In contrast to APM (Advanced Power Management, a predecessor of ACPI) [2], ACPI specifies the operating system (through OSPM) as the responsible entity for all power management decisions and actions using the interface defined by the specification. The available hardware is exposed to the operating system through a global ACPI namespace which not only describes the type of the hardware but also the type of power management facilities available for each device. Moreover, ACPI defines some system description tables used to specify special features (such as memory affinity of processors).

ACPI defines many different power states for the system and the different devices. *G0* to *G3* define the system's global power state where *G0* is the working state. *G1* through *G3* designate the different sleep and off states of the

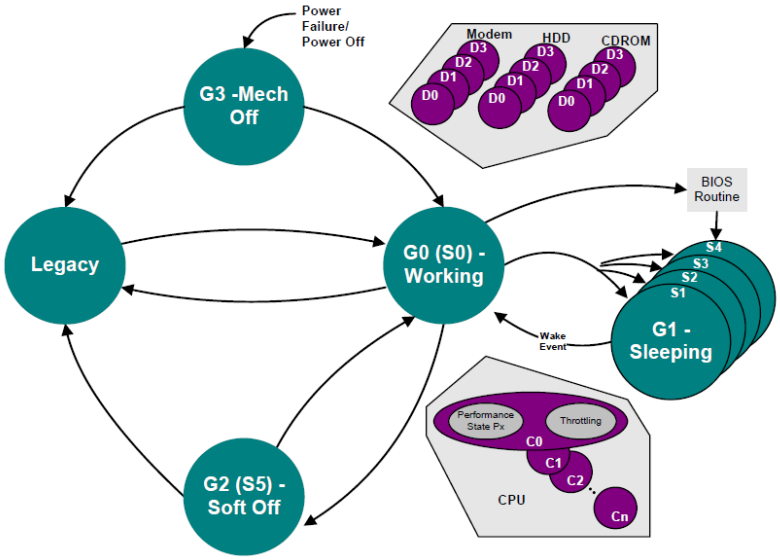


Figure 2.1: All system power states as defined by the ACPI specification [1].

system in which the system is not running (i.e. executing code). In the state *G0* the system's devices (e.g. modem, HDD, CD-ROM) can reside in a state between *D0* and *D3* where *D0* is the working state. Again, *D1* through *D3* designate the different sleep state in which the device is not operational. Similarly, the CPU can be in the running state *C0* or one of the sleep states *C1–Cn*. The individual sleep states differ from each other by their power consumption, by the time needed to change to and from the running state (i.e. latency) and by how a CPU enters the specific power state. Of course, these parameters are architecture specific and have to be provided by the processor manufacturer.

Different from other devices, the CPU has a second power management option. While in running state (Power State *C0*) the operating system can adjust the processor's voltage and frequency (dynamic voltage and frequency scaling — DVFS). For DVFS two different interfaces are present in ACPI, one to grant dynamic and continuous scaling and the other defining a set of performance states with well known performance and power ratios.

The interface to switch between the different CPU power states is defined by ACPI using ACPI registers. The registers are defined in the processor object declared in the ACPI namespace. For standard ACPI registers, a read to that register will put the processor in the desired state. If the register is defined as being *Functional Fixed Hardware*, the state transmission is to be handled by a manufacturer-provided CPU driver. Besides the interface for power management actions, ACPI also defines interfaces to get information on which to base power management decisions. For instance, when considering to put a processor in a sleep state, the operating system should take into account how much power will be saved and how long the state transition will take (i.e. how much potential computation time will be wasted once we want to schedule something

[1] Copied from the ACPI specification [1], page 27

on the CPU). ACPI provides the operating system with an interface to many such values describing the system. The values, however, still have to be provided by the manufacturers and at times the values might be unreliable, wrong or outdated. Anyway, for an operating system not to be bound to very specific hardware, these values provide the best way and modern operating systems like Linux are using them to improve their decisions on power management.

## 2.3 Processor Power Management Features

Modern processors and chipsets come with a multitude of features intended for power management. Some are hardware triggered and some are expected to be used by the operating system. As already mentioned in the last section, CPU power management can be split up into two dimensions. Voltage and frequency scaling on one hand and sleep states on the other hand.

### 2.3.1 Voltage and frequency scaling

Voltage and frequency scaling (DVFS) are often named together. While frequency impacts the performance and thus is uppermost in the user's perspective, power savings come mostly due to voltage scaling. On the other hand, voltage scaling does not (directly) impact performance. However, the frequency and the voltage of a processor are in a close relation which depends on the exact design of the chip. Meaning that every processor has a minimal voltage needed to support a certain frequency. Thus, a widely used approach is that the operating system (or, in general, the system's power-management component) sets the CPU to run at the desired frequency and the voltage is set as low as possible. Equally, the two available ACPI interfaces (throttling and performance states) do not differentiate between voltage and frequency scaling.

The power output of a processor is the sum of the static power and dynamic power. The static power consists primarily of various leakage currents. Dynamic power is a function of the core frequency and the core voltage and can be approximated by $C \cdot f \cdot V^2$, where C is the capacitance being switched per clock cycle, f is the core frequency and V is the core voltage. Obviously, a reduction in frequency will impact the power output only linearly while reducing the voltage will reduce power output quadratically.

### 2.3.2 Processor sleep states

A different approach to reducing power output is possible by letting the CPU sleep while there is no work to do. The ACPI specification tackles this approach defining different C-states for each core. The individual C-states differ in power consumption and latency as well as the method to put a core in the specific state. Modern processor implementations have widely adopted the C-states as defined by the ACPI specification.

*C0* is the running state, meaning that while in this state, the core is executing instructions.

The shallowest sleep state is *C1*. All processors have to support this sleep state in order to conform to the ACPI specification. The state is a special case as it is firstly entered by calling a native instruction of the processor (HLT for

IA-32 processors). Secondly, the latency to enter the state has to be low enough that the operating system does not consider the latency aspect of the state when deciding whether to use it. Modern desktop and server processors all support the *C1* state.

All deeper sleep states (*C2* and following) are optional and their adoption in today's processors is only partial. Obviously, manufacturers have focused on decreasing the power consumption of processors used in mobile computers and devices. Modern CPUs intended for use in mobile computers support sleep states as low as *C4*. When putting a core in a sleep state deeper than *C2*, the operating system has to make sure the core's cache is coherent once it resumes operation (a possible, simple solution is to flush the cache).

### 2.3.3 Observations for Multi-core processor

A special case is covered when dealing with multi-core processors. These processors combine multiple cores on one socket. Usually each core has some local, unshared cache (e.g. L1) and may share some higher level cache (e.g. L2 or L3). Additionally, depending on model details, processors might support individual voltage and frequency settings for each core or the voltage and frequency might be set only for all cores of a package at once.

#### Sleep states

Likewise, multi-core processors usually differentiate between per-core sleep states and processor-wide sleep states. For example, both AMD (Opteron) [6] and Intel (Xeon) [17] have introduced a proprietary C-state *C1E* which is hardware activated and puts the processor in a deeper sleep state *C1E* when all cores of that processor reside in sleep state *C1*, thus further reducing power output. This deeper sleep state is in both cases an extension of the *C1* state where voltage and frequency settings are set to the minimal possible value.

#### DVFS

When considering a NUMA (Non-Uniform Memory Architecture) multi-core processor, an additional performance consideration comes into play. For a NUMA core some memory is local and directly accessible and some might have to be accessed by probing another core's cache. If this other core is running at a reduced frequency to save power, such a cache probe will be slowed down as well, as the servicing core will serve the probe at the currently set frequency. Thus, using DVFS on a core might impact performance on other cores even if the architecture supports per-core voltage and frequency settings. If the other core is in one of its sleep states, the core will wake up momentarily to serve the cache probe and then return to the sleep state.

## 2.4 Barrelfish

As I will show and evaluate the result of this thesis using the Barrelfish operating system, I will give a short overview of its workings focusing on the parts more important to my work.

### 2.4.1 Overview

Barrelfish has been developed at ETH Zürich as an implementation of the newly proposed multikernel architecture [4]. The multikernel model includes the use of multiple independent operating system instances communicating via explicit message passing. In Barrelfish each OS instance is implemented as a vertically structured microkernel where the instance is factored into a kernel and a user-space part. Further functionality is given by user-space services and device drivers.

Barrelfish comes with a user-space library which provides helper functions for most of the available privileged features such as message passing or interrupt handling. The library functions act as wrapper functions for system calls, calls to the monitor interface or other special services.

### 2.4.2 Messaging system

Barrelfish differentiates between intra-core and inter-core communication. Both systems are accessible to user processes using the Barrelfish library which provides convenience functions for sending and marshalling messages.

Intra-core communication is handled by the kernel. To send a intra-core message the sender invokes the kernel and then the kernel delivers the message to the receiver and if necessary unblocks it.

For inter-core communication Barrelfish uses a variant of user-level RPC (URPC) [5]. Sender and receiver have to set up a shared memory region representing the channel between the two. In order to receive a message, the receiver has to periodically poll the channel for new messages.

In summary, intra-core communication is delivered by the kernel unblocking the receiver if necessary. Inter-core communication is processed completely within the user processes. When an inter-core message is sent a blocked receiver will not be unblocked.

### 2.4.3 Dispatcher

User-space processes consist of several dispatcher objects, one for each core the process should run on. Dispatchers provide an upcall interface invoked by the kernel to dispatch the process. Above this upcall interface the dispatcher runs a core-local user-level thread scheduler. Message handling and reception is also handled by the dispatcher as implemented by the Barrelfish library. Moreover, the threads package of the library provides an API for thread creation and termination as well as for thread synchronisation.

### 2.4.4 Monitor

The Barrelfish operating system includes a privileged user-space process called monitor. The monitor process is the user-space element for the operating system. In contrast to the kernel, the monitors on the individual cores communicate with each other. These communication channels are used for inter-monitor coordination and to give other user-space processes the ability to access other cores (e.g. to spawn a new dispatcher on a different core).

### 2.4.5 System knowledge base (SKB)

The Barrelfish operating system is built with a heterogeneous system in mind. In order for an operating system to be runnable on multiple different systems, the system has to be abstracted to let system services and primitives (like the messaging system) make use of present hardware characteristics. For this purpose, Barrelfish employs a special service known as the system knowledge base (SKB) [27]. The SKB is loaded with static and dynamic information about the system's architecture and characteristics.

The SKB is implemented using a constraint logic programming system called ECLiPSe [9]. The SKB is a system service serving as a wrapper to the ECLiPSe platform. An advantage of this combination is that it is possible to use constraint optimisation queries to gather information as needed about the system. Other processes and services interact with the SKB through the standard messaging channels given by Barrelfish.

## 2.5 ECLiPSe

ECLiPSe is the underlying framework of the SKB and thus the central mechanism used for optimisation in my thesis. In this section I give a short introduction to its workings and explain the basic terminology used in the paper.

ECLiPSe [9] is a constraint logic programming system. Logic constraints are written in a constraint-enhanced Prolog-compliant language. ECLiPSe ships with a multitude of libraries providing normal methods (e.g. list manipulation) and a set of libraries implementing the constraint logic.

The usual ECLiPSe optimisation application consists of a model of the system to be optimised and a cost function to calculate the cost of an individual system state. Minimisation is then applied using, for instance, a branch-and-bound method to identify the state yielding the minimal costs.

### 2.5.1 Language features

ECLiPSe provides a Prolog-like programming language including a collection of libraries. In this section I will give a short overview over the ECLiPSe programming language. For a full introduction or reference visit the ECLiPSe Website [9].

#### Terminology

ECLiPSe terminology is mainly borrowed from Prolog with a few addenda.

**Logical variables** Logical Variables are placeholders for values which are not yet known. In this they are similar to variables in other programming languages.

**Predicate** Predicates are the ECLiPSe equivalent to procedures and functions in other programming languages. The notation `pred/3` denotes a predicate named *pred* which has three parameters.

**Goal** A goal is a logical formula that has to be executed. This includes predicates which have to be satisfied. Borrowing from other programming languages, one could say that a predicate is the definition of a function and a goal is the execution of the function. An ECLiPSe program can consist of multiple goals which are combined using conjunctions or disjunctions.

**Query** The initial goal given by the user is called a query.

**ECLiPSe database** ECLiPSe holds all predicates which are currently asserted in a database. This database can be updated at runtime to add new values or remove old ones. Thus, it is possible to update parameters of the loaded ECLiPSe program using normal ECLiPSe queries.

### Execution scheme

ECLiPSe executes a program by trying to sequentially satisfy each goal that is part of the program. Whenever a disjunction is encountered, multiple execution paths are possible. ECLiPSe will choose one path and when it reaches a goal that can not be satisfied it will backtrack to the last choice made and, if possible, take one of the remaining paths. In this way an execution tree is constructed. If no more backtracking options are available, the query fails. If a path is found that satisfies all goals, the query succeeds.

# Chapter 3

# Literature Survey

## 3.1 Introduction

There is an abundance of research in the field of power management. Since the early nineties, different solutions to the problem of how to best make use of the different hardware features have been proposed. In general, research in the area targets high performance and low power consumption. This can be achieved by minimising the power/performance ratio. Most of the research on power management concentrated on exploiting DVFS on single-core and SMP multi-core machines, only more recent research has pursued the additional challenge presented by NUMA machines and deeper halt states.

The field of processor power management can be separated in several categories, each category having its unique challenges and encouraging different approaches.

## 3.2 Single-Core Power Management

Power management on a single core is particular as one does not have to worry about inter-core frequency or voltage dependencies. A single-core operating system usually has a set of tasks which are ready to run and has to decide in which order they are to be executed. Thus, one of the main research topics in single-core power management has been the effective use of DVFS to minimise the power/performance ratio of a system.

Research in the area of single-core power management, while not fully adaptable to today's multi-core systems, has put forward many algorithms in the area of power-aware scheduling of real-time and non-real-time systems which, lightly adjusted, have found entry into the multi core research.

A special case in the domain of single-core power management are real-time systems.

### 3.2.1 Real-time systems

Power management in real-time system profits from the knowledge inherent to real-time systems of each tasks deadline and runtime. Therefore, non-real-time systems can be seen as a generalisation of real-time systems.

As real-time systems have complete knowledge about the system, most research in terms of applying DVFS for power management has been made in this area. And the research result has then been adopted to be used in non-real-time systems as well. Hence, the research in real-time systems has significant impact on power management of non-real-time systems (which are the target of this work).

All approaches to integrate DVFS in a hard real-time scheduler need the runtimes of the tasks they are to schedule, either by prior-knowledge or runtime measurements (or both). The basic approach is then to choose for each task a voltage such that overall energy consumption is minimised and all tasks still meet their deadlines. This can be paraphrased as letting the systems run as slow as possible given a set of tasks, deadlines and execution times.

**Static schedule computation** A set of research in this areas uses the aforementioned advantage of knowing each task's runtime and expected deadline to statically minimise power consumption of a closed system. A first approach was developed by Hong et al. [13]. They used the tasks' parameters to solve the optimisation problem ignoring possible non-regular impacts on the tasks' performance. As with all closed systems, their research left little to no room for non-predictable interference such as user interaction. Okuma et al. [24] optimised the algorithm by splitting the optimisation task in two parts. In a first part the scheduler assigns each task a time slice on the assumption that the highest voltage setting is used (i.e. the scheduler decides on a task order). In the second part the scheduler optimises for power consumption by assigning each task a runtime voltage. Quan and Hu [26] used a different approach to the static solving for a closed system adopting earlier research. They thus reduced the computational cost of calculating the optimised schedule and frequency setting.

A drawback of the above solutions [13, 24, 26] is that their solution will use one frequency/voltage setting per task. Shin et al. [28] showed, that a lower power/performance ratio can be achieved by changing the frequency setting during a task's execution (named *intra-task voltage setting*). Their solution breaks up a program at its basic blocks and calculates an optimal schedule considering a distinct frequency/voltage setting for each basic block. Similarly, Azevedo et al. [3] suggest also a compiler-based approach which additionally considers power limits given by the user which the solution will not violate. Of course, these solutions only work as long as the system is the same as at compile time, as computation time will change if the system suffers a heavy overall load.

These papers [3, 13, 24, 26, 28] present highly specialised solutions to be used primarily in specific closed systems. Their solutions are based on the assumption that the system is well defined at each point in time and thus will not result in the same performance on a more general system. However, these papers made first steps into optimising the use of DVFS on a set of tasks in terms of power consumption. More important, the work on intra-task voltage setting [3, 28] showed the importance of task characteristics which are not constant over execution time (such as how memory bound a program is).

**Dynamic schedule computation** In contrast to the other real-time solutions, Hong et al. [14] and Zhu et al. [34] proposed dynamic scheduling solutions which work with no prior knowledge or pre-compilation of the tasks to be

executed. While Zhu et al. only consider periodic hard real-time tasks, Hong et al. manage sporadic tasks with unknown arrival time (which makes static scheduling impossible) as well. Hong et al. present two algorithms to dynamically compute the DVFS settings. Zhu et al. adapt a PID feedback controller to dynamically set the core frequency.

Both Zhu et al. [34] and Hong et al. [14] present new algorithms and approaches to the problem of power management using frequency and voltage scaling. These solution are not restricted to real-time system and have been adopted by some research on non-real-time systems.

**Soft real-time scheduling** All previously presented papers deal with hard real-time scheduling. Soft-real time systems are already close to common non-real-time systems and the solutions presented in this area are easily applied to them. In contrast to hard real-time scheduling a soft real-time scheduler does not have to make promises to hold a deadline. However, missing deadlines will decrease system availability/usability and therefore the scheduler should still aim for it. Pering et al. [25] present a scheduler which determines workloads empirically and schedules them according to these estimates to reach the deadlines. Of course, as these estimates may be wrong, deadlines might be missed (hence soft real-time). This is an example of how the work on hard real-time power management can be adapted and used in a more general system.

### 3.2.2 Non-real-time systems

Since in a non-real-time system one does not have the total knowledge given in a real-time system, total power consumption can not be computed beforehand. The operating system has to make decisions as to how it can safe power without losing too much performance dynamically. This results in trying to minimise the power/performance ratio, sometimes taking into account user preferences.

Similarly to real-time systems different approaches can be taken to facilitate the decision of frequency scaling. Some solutions make use of pre-runtime computation (e.g. solutions using a specialised compiler) while other solutions try to minimise power consumption using only runtime computation.

#### Prior-knowledge DVFS

Hsu et al. [15] used a special compiler to add information about characteristics of each block of a program which in turn can be used by the operating system to decide on the throttling factor. Hsu et al. augmented programs with information about the memory boundedness. The operating system then can throttle the CPU during a memory intensive computation where the processor would be mostly idle. Thus the system will reduce power consumption while losing only little in terms of performance. The blocks are computed at compile time and their memory characteristics added. The actual matching from memory boundedness to throttling factor is computed using a pre-computed table. Such a table is system specific and Hsu et al. generated it using profiling techniques.

Weissel and Bellosa [32] also consider task characteristics in their solution. They used hardware event counter to profile the system and to identify the counters which have most impact on power consumption. For their system the computation resulting in a memory-related and a performance-related counter.

At runtime, this profiling data is then used to set the frequency to the best setting for given counter readings. The event counter values are specific to a task and the profiling data has to be regenerated for each system.

Both papers [15, 32] show the impact a task's characteristic have on the optimal setting of the frequency. Both groups used tasks' memory characteristics to decide the throttling ratio. However, both papers also need pre-runtime profiling and in the case of Hsu et al. [15] even recompilation of the program to be run. These drawbacks put these solutions in a disadvantage in terms of practicability.

#### Runtime DVFS

Before any hardware support for frequency and voltage scaling was available Weiser et al. [31] approached the problem of power management. They proposed three algorithms to minimise their metric of million-instructions-per-joule (MIPJ). Their three algorithms set a foundation in the area of processor power management and are often referred to in later publications. Govil et al. [11] presented several flaws in one of the algorithms from Weiser. Govil proposed new improved version and compared it to the original version. Lacking appropriate hardware both Weiser and Govil used simulators to evaluate their algorithms.

Grunwald et al. [12] evaluated the algorithms proposed by Weiser et al. [31] on real hardware (Itsy Pocket Computer with a StrongARM SA-1100 CPU). Their test consisted of playing a film using the the algorithms from Weiser with different parameter settings. However, different to the solutions presented by Weiser et al. [31] on their simulator, the resulting power savings were minimal. The poor performance resulted because the algorithms kept oscillating between two frequency never using the optimal frequency between the two frequencies.

Like Hong and Zhu [14, 34], Varma et al. [30] applied the idea of a PID controller to power management. Unlike Hong and Zhu, Varma et al. did not consider a real-time system for their solution. Hence they don't use the PID controller to get the next throttling ratio by targeting the deadline of the task. Varma et al. use the PID controller to predict the workload in the next time unit. Using the predicted workload their system then adjusts the frequency setting accordingly. An implicit drawback of using a PID controller is its need for correct parametrisation. The extensive parametrisation of the algorithm results in the algorithm becoming tailored to a certain system. This is another disadvantage when compared to general-purpose algorithms such as the ones presented by Weiser et al. [31].

The idea of using the memory boundedness of a task was already used in publications by Hsu [15] and Weissel [32]. Isci et al. [19] use the same core assumption in their work (i.e. that memory boundedness is the predominant task characteristic to determine the use of DVFS). Different to Hsu and Weissel their solution needs no pre-runtime computation or compilation of the program. Their solution identifies the phases with distinct memory characteristics by matching selected event counters with earlier values. Each time a previous pattern is found, the power manager will assume that the load will develop as in the previous cases and set the core frequency accordingly. Of course, this history table will need some time first to fill up in order for reasonable decisions to be

taken. Still, the algorithm does not make use of pre-runtime knowledge and its computation is strictly online.

## 3.3 Multi-core Power Management

Multi-core system impose additional boundaries and challenges to power management. For instance, saving power using voltage scaling is usually only supported per-processor as opposed to per-core. Further considerations are needed on NUMA systems to deal with performance dependencies between cores (see section 2.3.3 for details).

Research has approached multi-core systems from different angles. A first step in the direction has been made by special architectures like the multiple clock domain processors used by Wu et al. [33]. Their system would allow individual voltage and frequency settings for different parts of the processor. For example, the integer processing core and the floating point processing core might be run at different clock rates. Such systems are similar to today's multi-core systems in as much as they too support multiple clock domains, adding complexity to the DVFS optimisation problem.

Like research on single-core power management, work with multi-core systems focused on using voltage and frequency scaling with different algorithms. Thread migration has mostly been used as a method to increase performance by exploiting tasks characteristics. For instance, two memory-bound tasks will result in better performance when run on cores using a different memory bus. Another use of thread migration on multi-core systems is found in work addressing thermal problems. Thread migrations can be a way to reduce local hot-spots by scheduling two compute-intensive tasks on separate cores. To the best of my knowledge, using thread migration to minimise power consumption has not been addressed.

### 3.3.1 DVFS for thermal problems

Modern processors often are equipped with automatic temperature control. When in danger of overheating they will stall execution autonomously to reduce heat output. Research in this area has therefore focused on keeping the system's core temperature below a certain threshold, or as low as possible. Powell et al. [10] used a combination of careful assignment of new threads to cores and thread migration to spread the heat production as evenly as possible through the system. A similar approach has been chosen by Merkel et al. [23]. They used a special metric to decide on the assignment of threads to cores. Donald and Martonosi [8] use a distributed DVFS algorithm to keep the core temperature below a targeted threshold.

To summarise, these papers applied different power management techniques in order to maximise performance. Some techniques may be applicable to minimising power consumption, such as the distributed DVFS algorithm presented by Donald and Martonosi [8]. However, mainly relevant is the distinct use of thread migration, even if not primarily to reduce power consumption.

### 3.3.2 Per-core DVFS

While current processors do not support full per-core DVFS but usually allow only one voltage setting per die, this constraint is rarely addressed by research. Therefore there is a set of papers tackling this special case and proposing algorithms applying voltage and frequency scaling to individual cores.

Kadayif et al. [21] focus on optimising power consumption of parallel computations. Their solution applies frequency scaling to cores trying to minimise slack time when different threads of the program have to synchronise. The solution does not reduce performance, as the thread with the longest run-time will always be scheduled at the maximal frequency. While the algorithm reduces power consumption in most cases, it will not sacrifice performance for greater power reductions.

Wu et al. [20] adapt a solution for multiple clock domain processors [33]. The main difference is, that they now have to consider parallel execution. As a consequence, they change their earlier DVFS algorithm into a distributed DVFS algorithm to be used on multi-core processors. The solution applies throttling independently by identifying which cores are running critical threads. Here, a critical thread is a thread upon which other threads are blocking and thus is a major performance barrier for the system. A different approach to a distributed DVFS algorithm is presented by Isci et al. [18]. The authors assign a local power manager to each core which implements independent, open-loop core-wide management actions. In addition, a global power manager is used to issue individual power modes to each local manager. The global power manager has the unique advantage to easily consider special constraint imposed by the system's architecture (such as non-uniform memory access).

### 3.3.3 Per-processor DVFS

Some of the most recent publications heed the architectural restriction of today's systems that do not allow voltage scaling to be applied to each core of a processor individually. Merkel and Bellosa [22] apply DVFS to a processor if all (or many) of its cores are running memory-bound threads. They reason that in this situation the cores will mostly be stalled due to congestion of the common memory bus and thus reducing the frequency will have little impact on their performance. Their solution uses this method in combination with careful thread to core assignment only as a last resort (i.e. if too many memory-bound threads are present). The algorithm is implemented and evaluated on a quad-core Intel processor running Linux. The evaluation of their DVFS scenario yields a decrease in the power/performance ratio in the best case and no change in the worst. While Merkel and Bellosa had mainly performance in mind (voltage scaling is applied only in very few cases), they present a solution working on current hardware to decrease the power/performance ratio.

Dhiman et al. [7] analysed the costs and benefits of deeper sleep states (down to ACPI state C6) against dynamic voltage and frequency scaling. The results reveal that it is advantageous to always run the processor at full speed and then switch to a deep sleep state rather than running the processor at a low frequency when its utilisation is low. However, I see some points of controversy. First,

the deeper sleep states referred to by the paper are far from being supported by a majority of consumer products. Therefore, the conclusion is not yet widely applicable. Second, the paper does not deal with the restriction that even on the latest hardware deeper sleep states (such as the used state C6) can not be entered by an individual core but only by all cores of a processor in unison.

Snowdon et al. [29] presented in their most recent paper a power management solution which applies DVFS using different event counters. Similar to Weissel and Bellosa [32] they calculated which event counter are most important on the system. Using pre-generated profiling data, their system then applied the best setting according to the data. A central difference to earlier work is the separation of the model and the policy. While the model is extremely sophisticated, it does not consider the dependence of the power consumption on thread/core assignment. The policy presented in the paper applies DVFS depending on the prediction of the model.

## 3.4 Summary

Past research in the area of power management has been focused on the use of frequency scaling. For early research which was based on single-core processors frequency scaling presented the only means of saving power without halting the execution. Frequency scaling for single-core systems has been thoroughly examined and many different solutions have been proposed.

Exploiting sleep states has until recently been straightforward. For instance on a single-core machine, if no thread is runnable the processor enters a sleep state. Apparently, optimising sleep state usage has not proven as effective as voltage scaling. Only with the improved support of deeper sleep states of recent processors has this additional dimension come into play.

Systems which consist of multiple multi-core processors are not widely spread and thus it is not surprising that research in that direction has not had the attention of others (like DVFS).

# Chapter 4

# Approach

## 4.1 Problem Statement

As presented in chapters 3 and 2, there are many possible approaches to power management. For my thesis I concentrated on a new option given by multi-core, multiprocessor systems (i.e. systems with multiple multi-core processors). Such systems get used more frequently as multi-core processors become widely available.

A particular property of a multi-core, multiprocessor system is the surplus in power savings if a whole processor is put to sleep compared to if single cores on different processors are put to sleep.

I therefore defined two different power states. A core resides in the *sleep* state when no thread is running on it. The second state is the *deep-sleep* state and can only be entered by all cores sharing a processor at once. Of course, the cores will only enter the deep-sleep state once they have no runnable threads. The two sleep states are defined by their power consumption and their wake-up latency. A system running a set of threads will show varying power consumption depending on which cores the threads are scheduled on. The power optimisation problem is to choose the cost-optimal system state (i.e. assigning threads to cores) from all possible system states (an example is given in the figure 4.1).

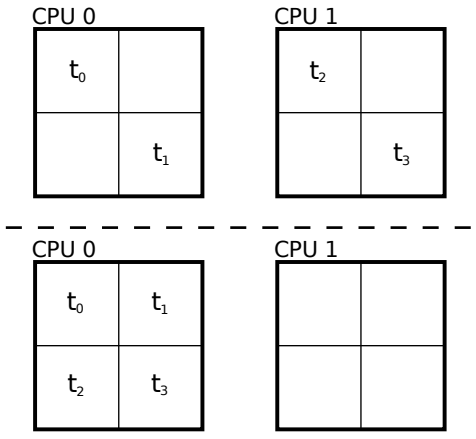


Figure 4.1: An example of two different threads to cores assignment. In the solution above, four cores reside in the sleep state. In the lower solution, four cores reside in the deep sleep state. Therefore the lower thread to core assignment is preferable (in most cases)

I consider only these two power states because between them they make up the difference of putting a processor of $x$ cores to sleep or $x$ cores on different processors. Nevertheless, the model is not bound by this definition. The cost model presented after the next section can easily be extended to include additional power states. In my model a power state is simply defined by its latency and its power consumption (which of course are architecture specific and thus parameters to the model).

## 4.2 General Approach

The solution to my problem will obviously include an optimisation of a system state in regard to several constraints. For this I use a constraint logic solving framework. The strength of such a framework lies in solving a problem where constraints can be easily added, removed or changed. Of course, the solution calculated by the constraint solving framework must then be applied to the system.

As a base for the logic constraints to be used by the framework, I constructed the formal cost model presented in the next section. The scope of the model is to calculate the cost of every system state. The power manager can then apply the state which generates the lowest cost to the system.

## 4.3 Formal Cost Model

I build the model bottom up, starting with a most idealised view and refining assumptions each step. At each step I state which assumptions have been removed and which new assumptions have been added in their place. A system state is defined by the assignment of threads to cores and I assume that the system may change its state after each (OS-defined) timeslice.

The cost of a state is a trade-off between power consumption and performance available to the user. The relation between power (energy consumption) and performance (amount of work done) is defined by the user.

Knowing the cost of each possible state, the system can choose the state with the least cost to run next.

### 4.3.1 Energy consumption

Cores draw different amounts of power while residing in different power states. The energy needed to run a next state is the energy needed while running in the next state (state cost) plus the energy needed to change from the current state to the next state (state change cost).

#### State cost

Defining the switch between states as instantaneous, I first draw up the cost of the system being in a given state.

**Basic model** In the basic model, the cost of running n threads is equal to the energy used by n cores during the state. The power consumption of the system in the various states is defined by the architecture.
**Assumptions**

+ each core has two states
  - running (consuming power)
  - sleeping (consuming no power)
+ all cores consume the same amount of power
+ each core can run at most one thread
+ state change is done instantaneously

$$E_s = l_{timeslice} \times nc_{running} \times P_{running}$$
$E_s$ : energy used by the system in the given state
$l_{timeslice}$ : duration of a timeslice in the system
$nc_{running}$ : number of cores (and/or threads) running
$P_{running}$ : power consumption of a running core

**Power states** As mentioned above, my model defines 3 different power states with different power consumptions for each core. However, at this level state change latency is still ignored.
**Assumptions**

+ each core has three states
  - running
  - sleep
  - deep sleep
+ the power consumption in the different states is the same for all cores

$$E_s = l_{timeslice}(nc_{running} \times P_{running} + nc_{sleep} \times P_{sleep} + nc_{deep\_sleep} \times P_{deep\_sleep})$$
$l_{timeslice}$ : duration of a timeslice in the system
$P_{running}$ : power consumption of a core when running
$P_{sleep}$ : power consumption of a core when sleeping
$P_{deep\_sleep}$ : power consumption of a core when in deep sleep
$nc_{running}$ : number of cores in running state
$nc_{sleep}$ : number of cores in sleep state
$nc_{deep\_sleep}$ : number of cores in deep sleep state

### State change cost

The model in the last section was used to calculate the cost of the system being in a given state. It is now adapted to calculate the cost of the system changing from the current state to the next state and then running in the next state.

As each core might reside longer in the state than only for the next timeslice, I divide the energy needed for changing the state by the number of timeslices for which the cores will reside in the state. In doing this not only when the state is changed, but at every timeslice, the state change cost is ammortized over the whole period the cores reside in this state.

**Assumptions**

- – state change is done instantaneously
- \+ state duration is the same for all cores
- \+ power state change is done instantaneously
- \+ thread migration is done instantaneously

$\Longrightarrow E = E_s + (E_{sc}/n_{timeslice})$

$E$ : total energy consumption of the next state

$E_s$ : energy consumption of the system in the new state (from subsection 4.3.1)

$E_{sc}$ : energy consumption during state change

$n_{timeslice}$ : number of timeslices the cores will stay in the state

**Power state change** Changing the core's or processor's power state is not instantaneous. As during the power state change the processor continues to draw power, the model has to incorporate the energy used during this delay.

**Assumptions**

- – power state change is done instantaneously
- \+ power consumption during state change is constant
- \+ power state changes are done sequentially
- \+ all cores have the same latencies for power changes
- \+ power state change latencies are symmetric (i.e. going to sleep has the same latency as waking up)

$$E_{sc} = l_{sc\_pow} \times P_{sc\_pow}$$
$$l_{sc\_pow} = nsc_{run\_sleep} \times L_{sleep} + nsc_{run\_dsleep} \times L_{dsleep} + nsc_{sleep\_run} \times L_{sleep} + nsc_{dsleep\_run} \times L_{dsleep}$$

$l_{sc\_pow}$ : the time needed for the power state change
$P_{sc\_pow}$ : power consumption during power state change
$nsc_{run\_sleep}$ : number of cores changing state from running to sleeping
$L_{sleep}$ : latency for power state change between running and sleeping
$nsc_{run\_dsleep}$ : number of cores changing state from running to deep sleeping
$L_{dsleep}$ : latency for power state change between running and deep sleeping
$nsc_{sleep\_run}$ : number of cores changing from sleeping to running
$nsc_{dsleep\_run}$ : number of cores changing from deep sleeping to running

**Thread migration** Possible solutions for the next state will not always keep the current thread/core arrangement and therefore thread migrations might be part of the system changing from one state to the next. As with power state changes, the cost of thread migration is the energy used until the migrations are over and normal operation can resume.
**Assumptions**

- − thread migration is done instantaneously
- \+ power state change and thread migration are done sequentially
- \+ the latency of thread migration is constant
- \+ power consumption during thread migration is constant
- \+ thread migrations are executed sequentially

$$E_{sc} = E_{sc\_pow} + E_{sc\_tm}$$
$$E_{sc\_pow} = l_{sc\_pow} \times P_{sc\_pow}$$
$$E_{sc\_tm} = n_{tm} \times L_{tm} \times P_{tm}$$

$E_{sc\_pow}$ : energy consumed by the system while changing power states
$E_{sc\_tm}$ : energy consumed by the system while migrating threads
$n_{tm}$ : number of thread migrations needed to enter the next state
$L_{tm}$ : latency of a thread migration
$P_{tm}$ : power consumption during thread migration

### 4.3.2 Work output

A counterpoint to the energy cost is the performance of the system. The performance of the system is equivalent to the amount of work processed by the system's computing cores. For example, if the system decides to run multiple threads on the same core in order to save power, it will do so at the cost of lost

performance. So, it is necessary to weigh energy consumption of a state against the amount of work the system will do during that state.

$\implies Cost_{tot} = E - \alpha \times W$

$Cost_{tot}$ : total cost of the next state (to be minimised)

$E$ : energy consumption of the system in the new state (from section 1)

$\alpha$ : user defined parameter to relate energy consumption and work output

$W$ : work output (i.e. performance) of the system during the next state

**Running multiple threads on one core**

Each core that is running (a thread) contributes to the system's performance. Thus, whenever multiple threads are scheduled to run on the same core, work output is diminished. As the system will stay in the next state only for the next timeslice, the running thread will not be preempted. Thus, the work output of each core depends only on the amount of work the core gets done per second and the length of each timeslice.

**Assumptions**

+ overheads due to thread switching for pre-emptive multitasking are ignored
+ all cores have the same, constant performance

$W = nc_{running} \times Perf \times l_{timeslice}$

$nc_{running}$ : number of cores running in the next state

$Perf$ : performance of a core (i.e. amount of work done by a core per second)

$l_{timeslice}$ : duration of a timeslice in the system

### 4.3.3 State definition

In order to relate the different variables to each other a state variable has to be defined. The state is denoted using superscript indices in all state dependent variables. In the current model state dependent variables are:

- $Cost^i_{tot}$
- $E^i, E^i_s, E^i_{sc}, E^i_{sc\_pow}, E^i_{sc\_tm}$
- $W^i$
- $nc^i_{running}, nc^i_{sleep}, nc^i_{deep\_sleep}$
- $nsc^i_{run\_sleep}, nsc^i_{run\_dsleep}, nsc^i_{sleep\_run}, nsc^i_{dsleep\_run}$
- $n^i_{tm}$

In order to define a state some new architecture-defined variables are needed:

- $C$ : set of all cores (system defined)
- $T$ : set of all threads (user defined)

A state is defined by a schedule $S^i$, defining which thread runs on which core:

- $S^i = \{(t, c) | t \in T \wedge c \in C \wedge$ 'thread $t$ runs on core $c$'$\}$

Based on the schedule some more state variables can be introduced:

- $C^i_{running} = \{c \in C : \exists t \in T((t, c) \in S^i)\}$
  (set of all cores where a thread is scheduled)
- $C^i_{deep\_sleep} = \{c \in C : c \notin C^i_{running} \wedge \forall c_1 \in C($ '$c_1$ and $c$ on same CPU' $\implies$ $c_1 \notin C^i_{running})\}$
  (set of all cores residing in the deep sleep state)
- $C^i_{sleep} = \{c \in C : c \notin C^i_{running} \wedge c \notin C_{deep\_sleep}\}$
  (set of all cores residing in the sleep state)
- $M^i = (S^i \setminus S^{i-1}) \setminus \{(t, c) \in S^i : \forall c_1 \in C((t, c_1) \notin S^{i-1})\}$
  (set of all thread migrations from state $S^{i-1}$ to state $S^i$)

### 4.3.4 Definition of variables

#### State-defined variables

Some variables can be defined using the sets defined in the last section. Effectively, these variables are functions of the state $S^i$.

- $nc^i_{running} = |C^i_{running}|$
- $nc^i_{sleep} = |C^i_{sleep}|$
- $nc^i_{deep\_sleep} = |C^i_{deep\_sleep}|$
- $nsc^i_{run\_sleep} = |\{c \in C : c \in C^{i-1}_{running} \wedge c \in C^i_{sleep}\}|$
- $nsc^i_{run\_dsleep} = |\{c \in C : c \in C^{i-1}_{running} \wedge c \in C^i_{deep\_sleep}\}|$
- $nsc^i_{sleep\_run} = |\{c \in C : c \in C^{i-1}_{sleep} \wedge c \in C^i_{running}\}|$
- $nsc^i_{dsleep\_run} = |\{c \in C : c \in C^{i-1}_{deep\_sleep} \wedge c \in C^i_{running}\}|$
- $n^i_{tm} = |M^i|$

#### System constants

Other constants are either system defined, measurement results or need to be approximated by what is known.

**State duration** The duration of each state is exactly a timeslice. The timeslice is given by the Operating System as the unit of time each thread can run at maximum.

**Power consumption constants** The power consumption of a core residing in the different power states ($P_{running}$, $P_{sleep}$, $P_{deep\_sleep}$) is usually made public by the processor manufacturer. Otherwise, the values may be gathered using a power sensing device either at system start-up, at run-time or beforehand.

The power consumption during certain special states (i.e. during power state change ($P_{sc\_pow}$) and during thread migration ($P_{tm}$)) need either be approximated or measured. As measurement of these values might prove difficult due to the short interval time and thus high demand to the sensor device, I assume them to be equal to the power consumption while running ($P_{running}$).

**Latency constants** The latencies of changing the power state ($L_{sleep}$, $L_{dsleep}$) consist of two parts. First, the latency of the actual power state change by the core or the CPU is usually published by the processor manufacturer. Second, the time needed for the operating system to halt or restart a single core has to be measured at start-up or run-time as it is dependent on architectural constants like number of cores in the system.

The latency of thread migrations ($L_{tm}$) depends on the operating system and the system architecture. Thus it needs to be measured — be it at start-up or at run-time.

#### Undefined variables

**State duration** In paragraph 4.3.1 I introduced the value $n_{timeslice}$ in order to amortise the cost of changing state over the whole state period. However, as I don't assume to have knowledge about thread lifetimes in my system, the value must be approximated.

**User parameter $\alpha$** As described in section 4.3.2, $\alpha$ is used to compare and weight the energy consumption to the performance of the system. Using $\alpha$ the user can choose the cost model to give more weight to the work output (leading to a more performant system) or to the energy consumption (leading to a more power economical system). The system constant $Perf$, introduced in paragraph 4.3.2, is a constant inherent to the system. As it is a highly abstract value, definition or measurement of this constant is non-trivial. Thus it is advantageous to eliminate it mathematically by defining a new user defined parameter $\beta = \alpha \times Perf$.

### 4.3.5 Conclusion

While the presented cost model is not in all parts an accurate description of reality, it is a fair approximation taking into account the most important factors of power consumption. Despite the many assumptions, the model has become increasingly complex. In the following, I list the complete model that has been extracted after all the refinement steps.

**Assumptions**

- core has three states
  - running
  - sleep
  - deep sleep
- the power consumption in the different states is the same for all cores
- thread migration is done instantaneously
- power consumption during state change is constant
- power state changes are done sequentially
- all cores have the same latencies for power changes
- power state change latencies are symmetric (i.e. going to sleep has the same latency as waking up)
- power state change and thread migration are done sequentially
- the latency of thread migration is constant
- power consumption during thread migration is constant
- thread migrations are executed sequentially
- overheads due to thread switching for pre-emptive multitasking are ignored
- all cores have the same, constant performance

$$
\begin{aligned}
Cost^i_{tot} &= E^i - \beta \times W^i \\
\\
E^i &= E^i_s + E^i_{sc} \\
E^i_s &= l_{timeslice}(nc^i_{running} \times P_{running} + nc^i_{sleep} \times P_{sleep} + nc^i_{deep\_sleep} \times P_{deep\_sleep}) \\
E^i_{sc} &= E^i_{sc\_pow} + E^i_{sc\_tm} \\
E^i_{sc\_pow} &= P_{sc\_pow} \times (nsc^i_{run\_sleep} \times L_{sleep} + nsc^i_{run\_dsleep} \times L_{dsleep} \\
&\qquad + nsc^i_{sleep\_run} \times L_{sleep} + nsc^i_{dsleep\_run} \times L_{dsleep}) \\
E^i_{sc\_tm} &= n^i_{tm} \times L_{tm} \times P_{tm} \\
\\
W^i &= nc^i_{running} \times l_{timeslice} \\
\beta &= \alpha \times Perf
\end{aligned}
$$

# Chapter 5

# Implementation

My implementation is described in two parts. In section 5.1 I describe the implementation of the cost model using ECLiPSe. Section 5.2 describes my solution to put cores to sleep and wake them up again. How I combine the two parts is described in section 5.3. Section 5.4 points out what has not been addressed by my implementation.

## 5.1 ECLiPSe Implementation

The formal cost model introduced in section 4.3 provides the formulae needed to calculate the different costs of a system and how they are combined. For a given state it will produce a corresponding cost value dependent on system and user parameters. To find the optimal state, I can evaluate the cost of every possible system state and then choose the state which yields the minimal cost. I chose to implement this using the constraint logic programming system ECLiPSe (see 2.5). Using its Prolog-like language, adding and removing constraints to build a complete model comes naturally. Moreover, ECLiPSe already provides libraries which can be used to minimise the cost of a model. For the optimisation I chose the branch and bound library. Its minimisation predicate will traverse the whole solution space returning the solution yielding the lowest cost (for an explanation of how ECLiPSe traverses the solution space see the paragraph *Execution scheme* in section 2.5).

The initial query to calculate the system uses the optimisation predicate given by the branch and bound library. This will find the state yielding the minimal cost out of all states satisfying the model. Of course, if no satisfying state is found the predicate will fail. The model itself consists of two parts: the predicates which generate the state to be evaluated, and the predicates which will evaluate the cost of that state using the formulae derived in the formal cost model in section 4.3. An advantage of the split implementation is that each part can, if necessary, easily be adjusted to new requirements without having to change everything.

### 5.1.1 State generation

As introduced in section 4.3.3, a state is defined as a set of thread/core pairs. These pairs represent thread to core mappings indicating for each thread on which core it is to be run. It is possible that multiple threads are mapped to the same core and that some cores have no threads mapped to them. If a core has no threads assigned to it, it will sleep.

In the ECLiPSe implementation the state is computed in a central predicate `save_next_state/1`. In the general implementation of the predicate any mapping of threads to cores which has every thread mapped to a core is a possible solution. Obviously, this leads to multiple possible solutions which span the solution space. From the state generated by this predicate, every other state variable declared in sections 4.3.4 and 4.3.3 of the formal cost model can be calculated. The state variables will then be used in the second part of the ECLiPSe implementation to calculate the cost of the generated state.

### 5.1.2 State evaluation

To evaluate a state and compute its cost, I added some code to the ECLiPSe program which transcribes the formal model from section 4.3. As the model consists solely of mathematical formulae and all needed variables have been calculated in the first step, there is a one to one mapping between the predicates of the ECLiPSe program and the formulae of the formal model. The state evaluation consists only of the cost formulae derived from the formal model and the model is not further constrained.

For example, the energy used during thread migration ($E_{sc\_tm}$) is defined in the formal model as:

$$E_{sc\_tm} = n_{tm} \times L_{tm} \times P_{tm}$$

$E_{sc\_pow}$ : energy consumed by the system while changing power states

$E_{sc\_tm}$ : energy consumed by the system while migrating threads

$n_{tm}$ : number of thread migrations needed to enter the next state

$L_{tm}$ : latency of a thread migration

$P_{tm}$ : power consumption during thread migration

The ECLiPSe implementation of this function is represented by the predicate `thread_migration/1`:

```
thread_migration(EnergyThreadMigration) :-
        getval(migrations, Migrations),
        NTM is length(Migrations),
        lat_thread_mig(LTM),
        power_thread_mig(PTM),
        EnergyThreadMigration is NTM * LTM * PTM.
```

The return value of the predicate `EnergyThreadMigration` is calculated exactly the same way as the $E_{sc\_tm}$ variable from the formal model. In section 4.3.4 I have referenced some variables which are either given by the system or by the user. All those variables are present in the ECLiPSe model as dynamic variables which must be set by the system before the first query.

### 5.1.3 Improving execution

The general implementation of the ECLiPSe model can be improved in different parts. Some improvements affect execution in general and some in individual cases.

#### Execution frequency

In the formal cost model (section 4.3) I assume the model is reevaluated after each timeslice. However, the solution retrieved by the model will stay constant for most of the time. Only the arrival of a new thread or the termination of a running thread can put the system in a state where its thread/core assignment is non-optimal in regard to the model's parameters. Thus, it suffices to query the model for a new solution only when a thread is created or a running thread terminates.

As a thread's affinity might change during its execution, this might seem to violate the possibility of extending the model to include thread affinity into the cost computation. As a solution, such a thread could be modelled as two threads, each with constant thread affinity.

#### Decreasing the solution space

The general `save_next_state/1` predicate will satisfy any assignment of threads to cores. This leads to a solution space with an exponential size on the order of $m^n$ where $m$ is the number of cores and $n$ is the number of threads. The ECLiPSe constraint solver will try each of these states. Most will not satisfy all following goals and many are redundant due to the symmetry of the system. The amount of unnecessarily evaluated states leads to a runtime of several minutes – even for comparatively easy cases.

For any practical use such a runtime for the optimisation is too high. Hence, I had to optimise the runtime of the ECLiPSe model. Since the whole solution space is traversed during minimisation, runtime can be decreased by restricting the solution space. If the individual goals of the model are more constrained (leaving less choice to the interpreter) the resulting solution space is much smaller and thus the solution will be found faster. While reducing the amount of states generated is an efficient optimisation, one has to make sure that the assumptions made to remove states from the solution space do not remove the optimal state as well.

I have implemented two different specialised versions of the general `save_next_state/1` predicate to handle two special cases. Both cases are optimised to handle a single change (either removal or addition of a thread). In the case of multiple changes the system will process them sequentially. If the number of pending changes is higher than the runtime gained, the general, non-optimised version can be queried. As the general version will examine every possible state, its runtime is not dependent on the number of removed or added threads.

**Adding a thread** The first optimisation treats the addition of a single thread to the system. If a single new thread has to be scheduled, there are only few sensible actions to be considered.

1. A new thread is assigned to any of the cores.
2. Possibly migrate threads which are sharing a core to a new core.
3. Exclude migrations forming chains from the above rule.

The first action is reasonable since a newly created thread has to be scheduled on a core. In order not to constrain the solution more than necessary, any core is a possible destination.

The second action covers the case where the benefit in work output is higher then the penalty in power consumption if new cores are activated. This can be the case if enough threads are available to wake up a whole processor which was sleeping. As the work output is linearly dependent on the number of cores running, it might exceed the increase in power consumption caused by waking up the processor (depending on the setting of the model parameter $\beta$). Different to the first action, the section action is optional and the solution might not include any newly activated cores.

The third action restricts the additional migrations considered by the second action (to activate new cores). The migrations considered by the second action include the case where a thread $t1$ is migrated to a core $c1$ and a second thread $t2$ is migrated from core $c1$ to core $c2$ (see figure 5.1). This case would be identical to the case where thread $t1$ is migrated to core $c2$. As more migrations are involved in the first case, the second case is sure to produce the lower cost (therefore, the first case needs not be considered).

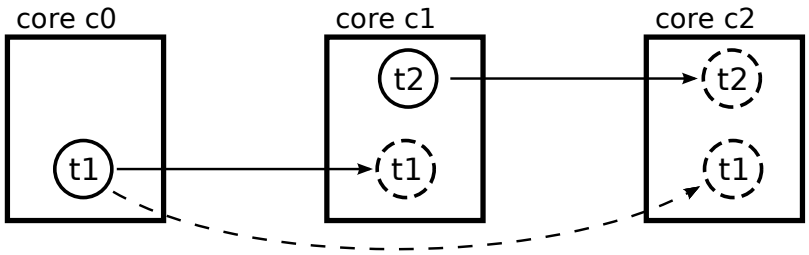


Figure 5.1: Migration chain — the striped migration will always yield a lower cost than the combination of the other two

This last restriction might not be applicable in a heterogeneous system where thread $t1$ might run on core $c1$ but, for limitations imposed by the system, not on core $c2$, whereas thread $t2$ can run on both cores.

These optimisations are quite simple and more is possible if one uses the symmetric property of the system at hand. However, my goal is to keep the implementation as general as possible. These optimisation sufficed to make the case of adding a thread usable in this work.

**Removing a thread** The second optimised case is the termination of a single thread in the system. To reduce the runtime of this case to an acceptable level, I had to leave behind much of the generality of the implementation. Similarly to the case of adding a thread, I have split up the solution space into a few actions.

1. Migrate one other thread to the core of the dead thread.
2. Migrate all threads sharing the processor with the dead thread to cores on other processors.
3. Keep the old state.

In contrast to the actions proposed in the case a thread had been added, all of these action are optional and exclusive, meaning that only one or none of the action is considered to generate the solution. The first action will most likely be considered if there is a core running multiple threads. In that case, migrating a thread to the dead thread's core might increase the work output without increasing the power consumption (in comparison to keeping the old state). The second action might be taken if moving all remaining threads from the dead thread's processor to other processors will save more energy than diminish the work output (not forgetting the $\beta$ factor, of course). Finally, the best state might be just the one the system was in before the thread died. In which case the third action will be taken and no migrations will be necessary.

These optimisation assume a lot in terms of symmetry and most probably would have to be replaced when facing a more heterogeneous system. However, the optimisations run well on today's hardware.

## 5.2 Barrelfish Implementation

I have extended the Barrelfish operating system to make use of the information available from the ECLiPSe model. The system knowledge base (SKB) is used to query the ECLiPSe program. The computed solution is then applied to threads of a single process.

### 5.2.1 Accessing the Barrelfish SKB

The Barrelfish SKB has two ways to be fed information. First, static files can be put in a special header file and can then be loaded by ECLiPSe just like a file (Barrelfish currently lacks a real file system, which is why the file has to be loaded in a header file). Second, the SKB has an asynchronous interface which will send queries directly to the ECLiPSe framework.

I use the static file to load my ECLiPSe program and then the asynchronous query interface to assert or retract state and parameter variables.

A set of wrapper functions is used to calculate a new solution and update the ECLiPSe database accordingly. Also, the ECLiPSe model is initialised to reflect the system with no running threads (i.e. the initial state). The wrapper functions return the solution of the query as the list of migrations needed to alter the current system state to the one returned by the ECLiPSe program.

### 5.2.2 Apply the system state

After application of the migrations calculated by the SKB, cores which have no thread running are supposed to halt. As Barrelfish doesn't yet support halting individual cores, I implemented the feature as a simple extension.

### Halting a core

In Barrelfish each core runs its own kernel of the operating system. The kernel automatically halts the core if the scheduler has no runnable user-space dispatcher. The status of a dispatcher (i.e. if it is runnable or not) can be set by the dispatcher itself and is used by the kernel when it decides which dispatcher to schedule next. So, if every dispatcher on a core sets its status accordingly, the core will halt.

Besides the dispatcher used for my implementation (see section 5.3) each core additionally runs its own monitor. The monitor is a privileged dispatcher providing part of the operating system interface to user processes. It is natural to let the monitor take a central role in the process of halting a core. In my implementation, the monitor provides an interface to other dispatchers which gives a dispatcher the possibility to register to the monitor as inactive. Once all dispatcher of a monitor's core are inactive, the monitor will set its own flag to not runnable. After that all dispatchers on the core will be not runnable and the kernel will halt the core. For this evaluation implementing the registration for only one dispatcher sufficed but the concept and the implementation can easily be extended.

### Waking a core up

Once a core has halted (and thus is sleeping), a mechanism is needed to wake the core up. A sleeping core will still handle any interrupt. To wake a core up an inter-processor interrupt (IPI) is sent to that core. Upon receipt of any interrupt the kernel will send a message to the dispatcher associated to it and schedule that dispatcher.

Again, an obvious choice to handle the IPI is the monitor dispatcher. Aside from being the last dispatcher on a core that becomes inactive, the monitor dispatcher is the first dispatcher on a sleeping core that is scheduled. In order to return all applications to an active state each dispatcher has to be activated (i.e. marked as runnable).

To achieve this, the IPI handler within the monitor sets its own status flag to runnable and then send an intra-core message to each of the dispatchers registered with the monitor as inactive. As explained in section 2.4, intra-core messages are delivered by the kernel. A dispatcher which has pending intra-core messages is scheduled without regard to its status flag. Thus, a intra-core message can be used to activate a dispatcher. Upon receipt of a monitor activation message, a dispatcher sets its status flag to runnable and continues normal operation.

Sending an IPI is a privileged operation and thus I added a system call with which the monitor can send an IPI to other cores. The feature is exported to all user processes as a monitor service to notify other cores.

### Message handling

The main problem with sleeping cores in an operating system like Barrelfish lies in the inter-core messaging system. As explained in section 2.4, inter-core messaging in Barrelfish is essentially writing to and reading from memory shared by the sender and the receiver. Every dispatcher keeps its own list of inter-core channels and polls for new messages each time it is scheduled. Therefore, if a

dispatcher is descheduled — which will happen when halting its core — it will not notice new messages in the shared memory channel.

Consequently, the sender of a inter-core message might have to activate the receiving dispatcher after sending the message. To activate the receiver, the sender can send an IPI to the receiving core. As discussed in the last section, the IPI will wake up the other core and activate all dispatchers running on that core. The receiving dispatcher will then handle the message waiting in the shared channel.

## 5.3 Combining ECLiPSe and Barrelfish

The previous sections describe how the solution is found (using the ECLiPSe model and the SKB) and how it can be applied, leaving this section to describe the part of the implementation which queries the SKB for a solution and executes thread migration and creation as necessary.

Since the SKB is already a central entity in the system, my implementation uses a single dispatcher to handle the communication with the SKB. In my implementation this dispatcher is called *coordinator*.

The coordinator is the main actor in my implementation. It initialises the model within the SKB, queries the SKB for a new solution when necessary, applies the solution to the system and updates the ECLiPSe model to reflect the new system state. The coordinator has an inter-core connection to a *client* dispatcher on each core. Using these connections, the coordinator coordinates clients to migrate a thread or start a thread. Also, the clients inform the coordinator before going inactive so that the coordinator knows at all times which cores are running and which are sleeping.

The implementation of the clients is quite simple. The client applies the commands (migrate a thread or create a thread) sent by the coordinator using the Barrelfish thread library. If the client has no more threads to run, it notifies the coordinator and then registers with the monitor as inactive. As in my setup the client is the only dispatcher besides the monitor, the monitor will make itself unrunnable and then the core halts.

The communication between the coordinator and the clients is strictly synchronous. Every message has an appropriate response (e.g. an acknowledgement) and the coordinator will wait for each response before sending the next message. Obviously, I preferred a simple implementation over performance or feature completeness beyond what I needed for my evaluation.

## 5.4 Limitations

As already mentioned my implementation neither claims performance optimality nor feature completeness. This section points out the different limitations of my implementation and how they affect the resulting solution.

### 5.4.1 Simplified implementation

In my implementation the solution from the SKB is processed by a separate dispatcher (the coordinator) and then sent to other cores (the clients) to apply

the solution. The main drawback is that only these dispatchers (the coordinator and clients) can create threads obstructing the possibility of Barrelfish to run multiple processes. An improved implementation should be part of the Barrelfish library, making the power manager ubiquitous and giving it global knowledge.

A second drawback in the Barrelfish implementation is the use of thread migration. As during my thesis Barrelfish was undergoing substantial changes, thread migration was not available. Instead, I simulate thread migration by killing a thread on the source core and creating it on the destination core. This does not impede the conclusions drawn in this paper, as the time it takes to complete a migration is system specific anyway (and a parameter to my cost model).

### 5.4.2 Possible optimisation

My implementation of the coordinator and especially the way it sends out the commands is as simple as possible. Its strictly synchronous implementation could gain much in terms of execution speed if changed to be more asynchronous. Performance was not central to my thesis and evidently my implementation will need more time to process the solution if many changes (i.e. migrations) are involved. However, my approach to query for a new state at thread creation or termination leads to the general case, where only few migrations are necessary. The disadvantage of a lower performance to update the system state to the new solution is therefore kept low.

### 5.4.3 Possible extensions to the model

For sake of practical applicability I had to severely optimise the Prolog model for a shorter runtime. To make these optimisations I had to exploit the symmetric architecture of today's processors. However, if a different solution for the runtime problem can be found, the cost model will prove easily extensible. The model might be extended to consider further sleep states, thread affinity to other threads or cores, cores yielding varying performance and more.

# Chapter 6

# Evaluation

As for most power management solutions, I evaluate my solution in terms of performance and power consumption. In section 6.1 I describe the system used for evaluation, including latency measurements of changing power state and migrating a thread as well as some preliminary power consumption measurements of the system. In section 6.2 I describe the exact procedure applied during evaluation, and finally in section 6.3 I present and discuss the results of the evaluation.

## 6.1 System

The ECLiPSe power model can be easily adapted for different machines. For my evaluation I choose an Intel Xeon system consisting of:

- Intel workstation board s5000XVN
- 2 quad-core 2.66GHz Xeon X5355 processors (see figure 6.1)
- 8GB of RAM

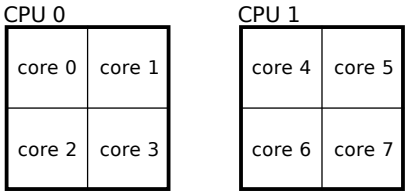


Figure 6.1: Processor and core layout of the system used for evaluation

### 6.1.1 Latency measurements

Apart from the layout of the cores and the processors, the ECLiPSe model needs latencies of power state changes and thread migration. These values are operating system and architecture dependent. I have measured the latencies using the HPET clock [16].

| | HPET ticks ($\sigma$) | ns ($\sigma$) |
|---|---|---|
| Migration | 107 (2.8) | 7473.0 (198.4) |

Table 6.1: Migration latency measurements

#### Latency of thread migration

The latency of thread migration depends heavily on how it is implemented. As explained in section 5.4.1 the user library of Barrelfish does not yet support thread migration. My implementation is a simple extension and implements only a limited version of thread migration. The measurements (table 6.1) depict the whole time span from the message sent to the first client by the coordinator until the second client confirms successful migration. Of course, the latency for thread migration increases with the system's load.

#### Core state change latency

| $\mu s$ ($\sigma$) | Halt | Wakeup | State change |
|---|---|---|---|
| State Change | 6004.5 (35.76) | 6182.8 (35.20) | 6093.7 (95.96) |

Table 6.2: State change latency measurements

The latency to halt or wakeup a core does not only encompass the time needed for the core to change its state. Halting a core as presented in section 5.2.2 of the implementation is more complex than that. Therefore, the latency must include the full time span from the moment the decision to change the state of a core is taken to the moment the state has actually been changed.

The latency of halting a core includes the time needed to inform the coordinator and registering with the monitor. After both dispatchers are inactive, still some more time will be spent by the system until the kernel executes the actual halt.

The time span to wake a core up starts when an inter-core message is to be sent to a sleeping core. The coordinator will instruct the monitor to send an IPI to the other core. Only when the receiving dispatcher is scheduled once more and handling the message is the wakeup sequence complete.

Table 6.2 shows the resulting measurement data. The difference between waking a core up and halting a core is most likely due to the IPI which is needed to wake a core up. Also the latency needed by the hardware as specified by Intel is not significant if related to the latency caused by the software. My ECLiPSe model uses the same value for wakeup and halt latency. For a system with a higher difference between the two latencies, the model can be easily adapted.

### 6.1.2 Power measurements

During the evaluation I used a standard watt meter which returned measurements with a period of one second. The power measurements are taken at the computer's power outlet. As I am using a normal off-the-shelf system and

measuring the power consumption at the computer's main power outlet, the processor is not the only power consumer in the system. Thus it is to be expected that the measurements taken will not be constant but have a certain variance due to the influence of other system components.

To show the impact the cores' power states have on the systems' total power consumption I have made some preliminary power measurements at different system states.

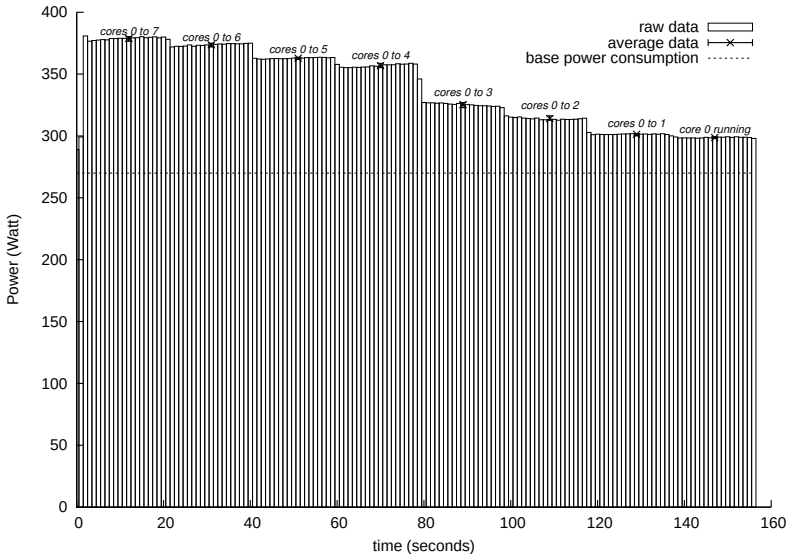


Figure 6.2: Halting the systems' cores sequentially

Figure 6.2 shows the measurements taken while sequentially halting the individual cores of the system. The base power consumption depicts the system's power consumption when all cores are powered down. As one would suspect, the biggest drop in power consumption is after halting core 4. Because cores 4 to 7 share a core, after halting core 4 all cores of the processor are halted and thus can transition into deep sleep. Notable as well is that the remaining drops are not as constant as the specifications suggest. The high drops after halting core 6 and after halting core 2 is explained by the fact that the Intel Xeon quad-core processor is constructed using two dual-core units. Thus, after halting core 6 and after halting core 2 the benefit is higher than in the normal case because the whole unit is sleeping. This is similar to what is observed when the whole processor is sleeping. However, using the measurement tools at hand, the actual difference is difficult to quantify. Thus and in order to keep the model general, my parameters for the model stick to the values given by Intel.

I already mentioned that a certain variance is to be expected in all power measurements. Caused by the power consumption of other system components than the processor. To visualise the variance, figure 6.3 depicts the power consumption of the system while having all processors running for a longer period. It can be seen, that even though the system is stable the power consumptions varies by little more than 10 Watts. The variance is still smaller than the power

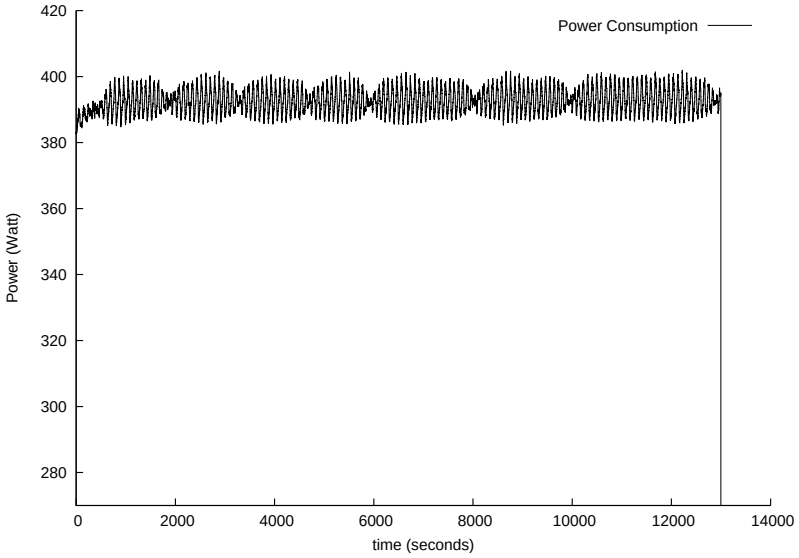


Figure 6.3: Power consumption while all processors are running

savings possible by halting cores and thus no impediment to my evaluation.

To show the power savings possible by optimising the assignment of threads to cores, I have measured the system's power consumption for different setups where four cores are running. Table 6.3 shows the data. The table shows the four possible setups when ignoring symmetrical possibilities. The measurements show that the maximal power savings when running four threads are about 25 Watt.

| | Cores running | Watt ($\sigma$) |
|---|---|---|
| Best case | 0,1,2,3 | 319.9 (1.2) |
| | 0,1,2,4 | 337.2 (0.4) |
| | 0,1,4,5 | 337.1 (0.4) |
| Worst Case | 0,2,4,6 | 346.1 (0.4) |

Table 6.3: Power consumption of different 4-core setups

## 6.2 Evaluation procedure

To evaluate my implementation I use compute-bound workload threads which will start and terminate at pre-defined points in time. As the workload I use a simple prime-number generator. I chose a prime generator for it is obviously compute-bound and can easily be migrated.

As a comparison a trivial thread-placement algorithm *nextFree* is considered, where new threads are simply scheduled on the next free core. My implementation is tested with different settings of the parameter $\beta$ *myBetaXX*. As a reminder, the $\beta$ parameter represents the weight of the performance. A higher

$\beta$ results in a solution with higher performance whereas a lower $\beta$ results in lower power consumption.

The measurements are taken using four different evaluation sequences. An evaluation sequence defines when each workload thread is created and when it terminates. Of course, the implementation of the cost model has no notion of when a newly created thread will terminate.

The first evaluation sequence *constant* is the trivial case, where eight threads are started and kept running for a long time. Thus, when applied to this sequence the different algorithms will change the state only at the start. After all eight threads are running, the system will keep a constant state not effectuating any migrations.

The *exploit* evaluation sequence is constructed to exploit the peculiarity of my implementation that it re-evaluates the system state after threads terminate.

The *arbitrary* sequence has been arbitrarily put together and uses a total of 19 threads. The *arbitrary* sequence has no long spans where no threads are created or removed because during such periods the system state would not change.

As a last variation, I use the *arbitrary* sequence but with much shorter intervals between the events (thread creation and termination).

## 6.3 Results

### 6.3.1 Evaluation sequence *constant*

The *constant* evaluation sequence simply starts eight threads and keeps them running for 165 seconds. I've evaluated the sequence with $\beta$ settings of 25, 50, 85, 100, 200, 300 and the *nextFree* algorithm.

The graph in figure 6.4 displays the measurements. The system's performance is calculated by averaging the performance of all threads of the system. The energy is simply measured using a common power meter.

The line *Base power consumption* at 1100 Wmin is the power that the system would have consumed if all eight cores were halted for the time of the evaluation sequence. Thus it represents the theoretical lower bound of what is possible in terms of energy consumption while applying processor power management. One has to consider that in order to reach this theoretical lower bound in power consumption the system's performance would drop to zero. Thus it is important to notice that the range possible for evaluation in terms of power consumption is bound by the system's base power consumption, while the performance's range can drop to 0.

The goal of power management is to minimise power consumption while maximising the system's performance. Thus an upper bound for performance is given in this evaluation by the *nextFree* algorithm. The *nextFree* algorithm needs effectively no computation and by design never does migrations. Therefore, while using the *nextFree* algorithm the system does not spend additional time querying a model or applying new states besides what is necessary for thread creation.

The two algorithms *myBeta300* and *nextFree* behave exactly the same in terms of which state was chosen. As the two algorithms generate the same set of system states, their power consumption is equal. Performance of *myBeta300*

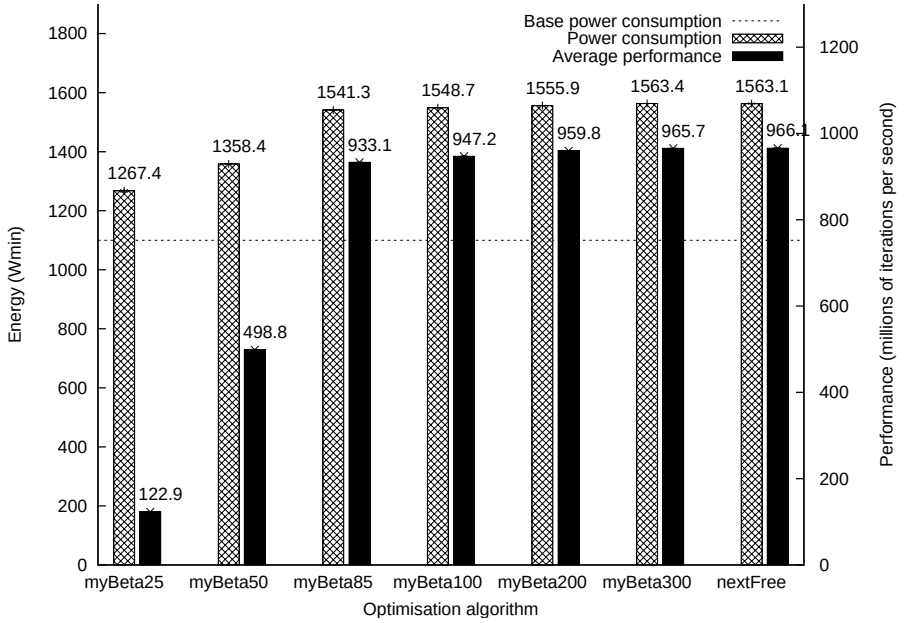


Figure 6.4: Power and performance values running *constant*

is slightly lower which can be explained by the additional computation needed by *myBeta300* to query the ECLiPSe model.

The graph shows how power consumption and performance decreases as the $\beta$ parameter is decreased. As each new thread is started, the ECLiPSe model compares the benefit in work output if the thread is put on a new core to the cost of activating that core from its sleeping state. In the case where the core is part of a still sleeping processor (i.e. all its cores reside in deep sleep), the cost of activating that core is much higher (due to the higher power savings when a core is in deep sleep). This comparison is biased using the $\beta$ parameter.

The algorithms *myBeta85, myBeta100* and *myBeta200* show only a slight decrease. Their solution differs as to when exactly the second processor is activated. In each case, the first four threads are assigned each to an individual core (thread 0 on core 0 etc.). Due to the lower $\beta$ setting (compared to *myBeta300*), the fifth thread is not assigned to a core on the second processor but kept on the first processor. As each new thread starts, the cost of activating the second processor is compared to the benefit of the additional work output if all threads were to run on their own core. For *myBeta200* this is the case when the sixth thread starts, for *myBeta100* after the seventh thread starts and for *myBeta85* after the eighth thread starts. Once all eight threads have been started, these algorithms all have each thread running on an individual core.

The algorithms *myBeta25* and *myBeta50* use such a low setting that the ECLiPSe model will never consider to activate the second processor. Thus in both cases all eight threads run on the first four cores for the whole length of the sequence. Thus, the steeper decline in power consumption and performance. Using *myBeta25* only the first core was running all eight threads. As the graph

shows, the system's performance with *myBeta25* is one-eighth of the system's performance with *nextFree*.

### 6.3.2 Evaluation sequence *exploit*

The *exploit* evaluation sequence (shown in figure 6.5) is designed specifically in favour of my solution. It assumes the usual approach would not recompute the thread to core assignment when a thread terminates. To express this, I compare the two algorithms *myBeta300* and *nextFree* which displayed similar power and performance numbers in the last evaluation sequence. *NextFree* is implemented with the mentioned flaw that it does not rearrange the threads when a thread terminates. For both algorithms, the eight threads will initially be scheduled on individual cores (i.e. thread 0 on core 0 etc.). The difference lies in what happens after threads 1,3,5,7 terminate.

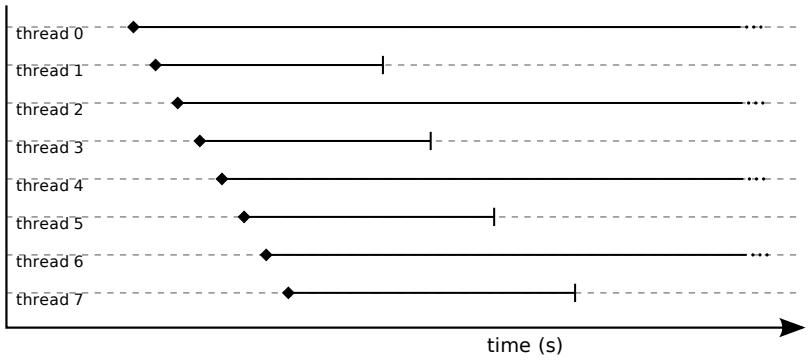


Figure 6.5: The *exploit* evaluation sequence

The *nextFree* algorithm simply ignores thread terminations and lets the remaining four threads run on cores 0,2,4,6. Looking back at table 6.3, this is the worst case possible for having four cores running.

Conversely, the *myBeta300* algorithm re-evaluates the system state after each thread termination and thus migrates the remaining threads (0,2,4,6) all onto the same processor.

The graph in figure 6.6 displays the measurements taken for the two algorithms. As was to be expected, both algorithms yield close to the same system performance. The two different core setups are equal in terms of performance. However, *myBeta300* has a lower power consumption. The difference is about 50 Wmin. From table 6.3 it can be seen, that the power consumption difference of the two core setups (0,1,2,3 vs. 0,2,4,6) is about 25 W. Those 25 W added over the sequence's time span yield the 50 Wmin.

The possible energy savings seem small in comparison to the total energy consumption. However, my solution optimises the use of processors only and saving 50 Wmin corresponds to saving 16% of the theoretical possible power savings.

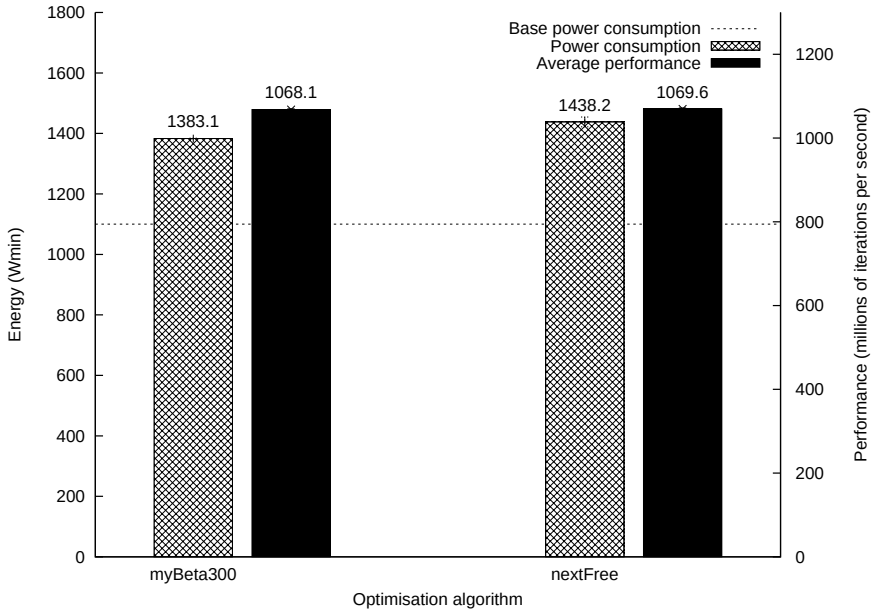


Figure 6.6: Power and performance values running *exploit*

### 6.3.3 Evaluation sequence *arbitrary*

The graph in figure 6.7 shows the power consumption and the performance of the system while running the *general* evaluation sequence (Figure 6.8) and using the different algorithms for thread placement.

Comparing my solution using different parameters (*myBeta50, myBeta100* and *myBeta300*) additional power savings are observable for lower settings of $\beta$. As already seen in the graph for the *constant* sequence (figure 6.4), the drop in performance is much more severe. A possible reason for this is that adjusting a system's overall performance by running multiple threads on the same core does not allow fine-grained tuning. Each time two threads are put on the same core their performance will effectively be halved. *MyBeta50* focuses on minimising power consumption in such an extreme way that the second processor is never powered up. Thus, all threads are running on the first four cores leading to a massive reduction in performance.

A $\beta$ setting of 300 favours performance at the expense of power consumption. For the given sequence the model will never result in a system state where multiple threads are sharing a core. The algorithm *myBeta300* performs nearly as well as the upper bound in performance given by *nextFree*. However, *myBeta300* does significantly better in terms of power consumption.

In contrast, the *myBeta50* favours lower power consumption at the expense of lower performance. Comparing it to *NextFree*, one has to keep in mind that, due to the system's base power consumption, the maximal possible energy saving would be 399 Wmin while the performance's range can reach zero. Thus the maximal possible loss in performance for the *nextFree* algorithm would be 913 million iterations per second. In view of this, a drop in energy from 1499

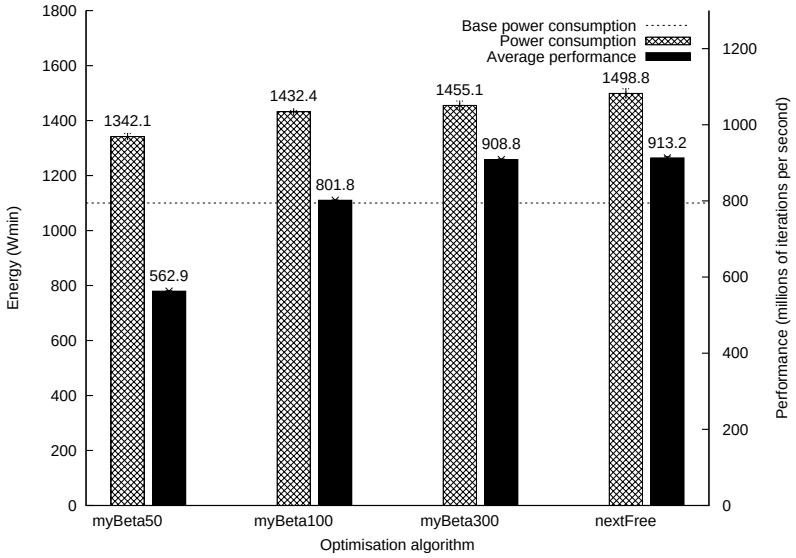


Figure 6.7: Power consumption and Performance running *arbitrary*

Wmin (*nextFree*) to 1342 Wmin (*myBeta50*) corresponds to a drop of 39% of the possible energy savings. While the drop in performance from 913 million iterations per second(*nextFree*) to 563 million corresponds to a drop of 38% of the possible performance loss. Therefore, if put in relation to what is possible, the changes in power and performance correlate better than the graph would suggest.

### 6.3.4 Evaluation sequence *overload*

The last evaluation sequence uses the same order of events as the *general* evaluation sequence. However, in order to evaluate the algorithm's performance under stress, the time spans between events (creation and termination of threads) is reduced below thread creation latency. Thus for all algorithms, a queue of events will build up which has to be processed. While the latency of the *nextFree* algorithm is bound by the latency of thread creation, the *myBeta300* algorithm needs additional time to query the ECLiPSe model. At three points during the sequence, the time between events is longer to let the algorithms to catch up.

The graph in figure 6.9 shows the measurements for the two algorithms *myBeta300* and *nextFree*. The *overload* evaluation sequence is much shorter than the other sequences leading to lower total energy consumption. As already shown, longer computation time does not significantly impact the performance, and this remains true even under stress.

The energy saving of *myBeta300* relative to the lower bound are for *overload* 9.4% whereas for *general* it results to 10.75%. While *myBeta300* performs slightly worse under stress in terms of energy saving, the difference is not significant.

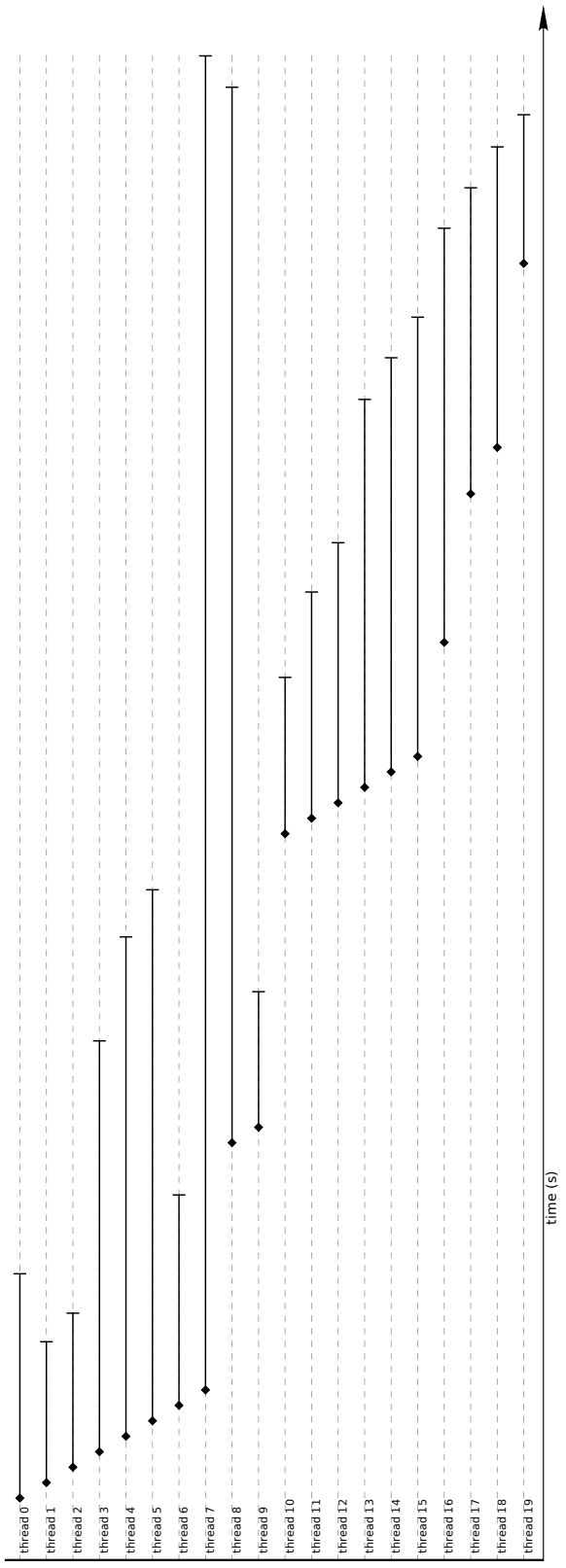


Figure 6.8: The *arbitrary* evaluation sequence

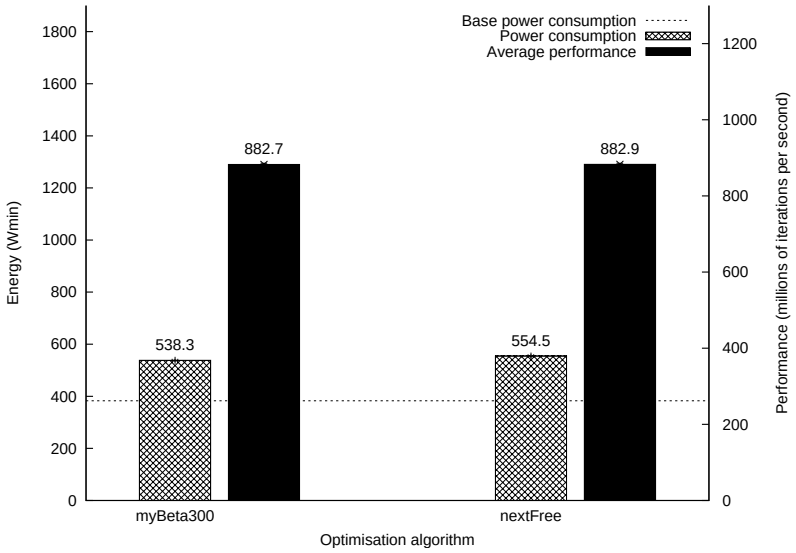


Figure 6.9: Power and performance values running *overload*

### 6.3.5 Cost of Optimisation

The *nextFree* algorithm is straight forward and thus a new state is easily computed. In contrast, my implementation will query the ECLiPSe model after each event. The total amount of time needed for the optimisation during the *arbitrary* evaluation sequence is shown in the graph in figure 6.10. As presented in section 5.1.3, I have separately optimised the model's execution when a thread is added and when a thread terminates. Therefore, the graph distinguishes between these two cases. The *myBeta50* algorithm used over 8 seconds of the 254 second evaluation sequence to calculate the optimisation. As would be expected, the *nextFree* algorithm needs close to no runtime to generate its solution.

To illustrate the impact the additional computation has on power consumption and performance I compared my algorithm *myBeta300* with a static version of it *mystaticBeta300* which has the solution pre-computed. Thus, the *mystaticBeta300* algorithm needs almost no time for the optimisation (like *nextFree*). Table 6.4 shows the difference in power consumption and performance while running the *arbitrary* evaluation sequence between *mystaticBeta300* and *myBeta300*. The table shows that *mystaticBeta300* does better in performance

| | Absolute Diff. | Relative Diff. |
|---|---|---|
| Energy (Wmin) | 7.5 | 0.5% |
| Performance (millions of iterations per second) | 4.2 | 0.5% |

Table 6.4: Difference of power and performance values of the *mystaticBeta300* and the *myBeta300* algorithm (running *arbitrary*)

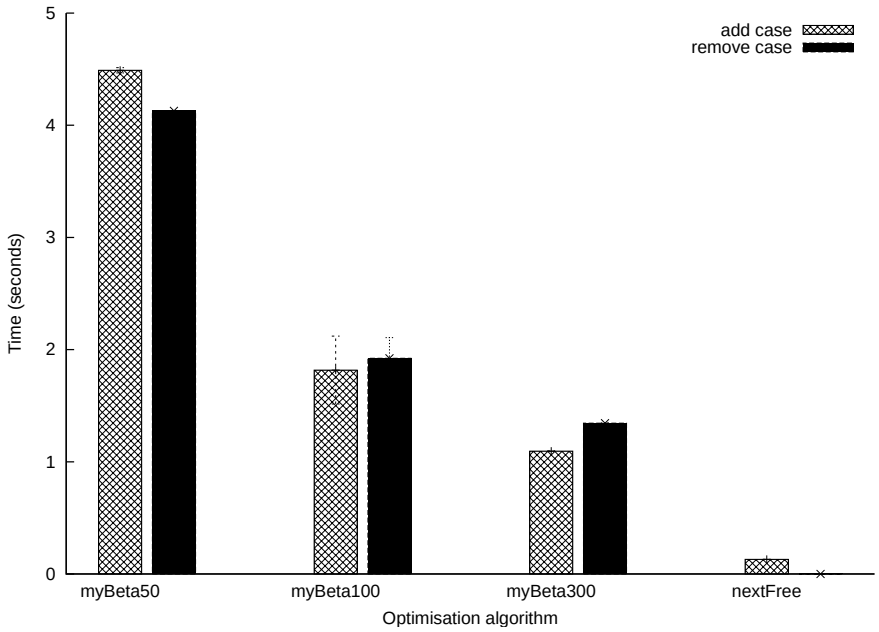


Figure 6.10: Total runtime of ECLiPSe optimisation during the *arbitrary* evaluation sequence

and power consumption. However, in relation to the absolute values from figure 6.7 the difference is only 0.5% in power and performance.

The *arbitrary* evaluation sequence has a high rate of changes in the system, leading to more queries to the ECLiPSe model and thus longer total time spent during optimisation. The additional computation does not go unnoticed in the efficiency of the system but in relation to the absolute values are not highly significant to the overall conclusion.

# Chapter 7

# Conclusion

In this thesis I presented a new technique for managing the power consumption of processors aside from the already well researched frequency scaling approach. My implementation on the Barrelfish operating system showed the opportunity in power management offered by intelligent thread to core assignment. My solution is easily parametrised as the needed values are either given by the manufacturer or can be measured at runtime. The model's main parameter $\beta$ gives the user or the operating system a possibility to influence the behaviour in favour of higher performance or lower power consumption.

If the highest setting for the model's parameter is used, performance is close to optimal while power consumption is lower compared to a trivial algorithm. Of course, the implementation is tightly coupled with the inherent characteristics of Barrelfish as a multikernel operating system. Nevertheless, the approach and the cost model are not operating system specific can can be used just the same on other systems using other parameters.

However, the measurement also showed how small the impact of power management on processor level is compared to the power consumed by the whole system. Obviously, applying power management to the processors should only be a part of a system's whole power management facilities. In relation to what can theoretically be achieved, my measurements reveal significant power savings while the system's performance can be influenced with the model's parameter.

## 7.1 Future Work

My implementation shows a set of problems which make it difficult to apply to more complex systems. For instance, the runtime of the ECLiPSe optimisation is a problem if a more complex system is considered. A possible solution could be found if keeping the system temporarily in a non-optimal system state would be considered. Then the optimisation could be performed and applied less frequently in order to keep total optimisation time low. Also, the integration with the operating system should be more thorough. For example, my implementation restricts the optimisation to threads of a single process.

Besides the correction of the aforementioned problems, many other possible extension and refinement points are left open. The core piece of my solution, the model, is constructed in such a way that further constraints to support different

architectures can be added without much effort.

For example, my model possesses a simple view on performance. In reality, a core's performance is not simply a linear function of the number of threads running on it. How well a thread performs while running on a core depends on its characteristics and the characteristics of the other cores scheduled on that core. For instance, two threads which are accessing the same physical memory will perform better if scheduled on the same core or cores sharing a first-level cache. Obviously the cost and benefits to performance deserve a more accurate model than given by the current implementation.

A further extension point to the model is its parametrisation. In most systems, individual threads have different priorities. Including these priorities into the model, highly prioritised threads could be assigned their own core for maximum performance while other threads can share cores to reduce power consumption.

Finally, the model could be extended to encompass more than just the system's processor by including power and performance costs of other system devices like the hard drive.

Aside from the cost model, the Barrelfish implementation could gain from a better integration with the system. Part of this is the currently imposed restriction that only the threads of a single process are monitored and optimised. Using the user-space library and maybe a new system service, a power management solution can utilise full knowledge of the systems state, and optimise thread placement as presented. Additionally, the implementation could adjust the model's different parameters (such as migration latency) on-line according to runtime measurements.

It was not the goal of this thesis to perform a complete implementation of the approach but rather to show the possible effectiveness of the approach in view of coming computer systems with multiple multi-core processors.

# Bibliography


[1] *Advanced Configuration and Power Interface*, October 2006. https://www.acpi.info.

[2] *Advanced Power Management v. 1.2 specification*, February 1996. http://microsoft.com/whdc/archive/amp_12.mspx.

[3] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 168, Washington, DC, USA, 2002. IEEE Computer Society.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[5] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, 1991.

[6] *BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors*, revision 3.08 edition, July 2007. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf.

[7] G. Dhiman, K. K. Pusukuri, and T. Rosing. Analysis of dynamic voltage scaling for system level energy management. In *HotPower'08: Proceedings of the 1st USENIX workshop on power aware computing and systems*. USENIX Association, 2008.

[8] J. Donald and M. 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.

[9] ECLiPSe. The ECLiPSe Constraint Programming System. http://www.eclipse-clp.org/. [Online; accessed 30-July-2009].

[10] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 260–270, New York, NY, USA, 2004. ACM.

[11] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM.

[12] D. Grunwald, C. B. Morrey, III, P. Levis, M. Neufeld, and K. I. Farkas. Policies for dynamic clock scheduling. In *OSDI'00: Proceedings of the 4th Symposium on Operating System Design & Implementation*, page 6, Berkeley, CA, USA, 2000. USENIX Association.

[13] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 176–181, New York, NY, USA, 1998. ACM.

[14] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 653–656, New York, NY, USA, 1998. ACM.

[15] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. *SIGPLAN Not.*, 38(5):38–48, 2003.

[16] *IA-PC HPET (High Precision Event Timers)*, October 2004. http://www.intel.com/hardwaredesign/hpetspec_1.pdf/.

[17] *Intel Xeon Processor 5300 Series Datasheet*, September 2007. http://www.intel.com/p/en_US/products/server/processor/xeon5000/technical-documents.

[18] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[19] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society.

[20] P. Juang, Q. Wu, L.-S. Peh, M. Martonosi, and D. W. Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 127–130, New York, NY, USA, 2005. ACM.

[21] I. Kadayif, M. Kandemir, and I. Kolcu. Exploiting processor workload heterogeneity for reducing energy consumption in chip multiprocessors. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 1158, Washington, DC, USA, 2004. IEEE Computer Society.

[22] A. Merkel and F. Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *HotPower'08: Proceedings of the 1st USENIX workshop on power aware computing and systems*. USENIX Association, 2008.

[23] A. Merkel and F. Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 1–12, New York, NY, USA, 2008. ACM.

[24] T. Okuma, T. Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In *ISSS '99: Proceedings of the 12th international symposium on System synthesis*, page 24, Washington, DC, USA, 1999. IEEE Computer Society.

[25] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the IpARM microprocessor system. In *ISLPED '00: Proceedings of the 2000 international symposium on Low-Power Electronics and Design*, pages 96–101, New York, NY, USA, 2000. ACM.

[26] G. Quan and X. S. Hu. Minimal energy fixed-priority scheduling for variable voltage processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(8):1062–1071, Aug. 2003.

[27] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

[28] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 438–443, New York, NY, USA, 2001. ACM.

[29] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: a platform for OS-level power management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302. ACM, 2009.

[30] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and J. Bruce. A control-theoretic approach to dynamic voltage scheduling. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 255–266, New York, NY, USA, 2003. ACM.

[31] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[32] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *CASES '02: Proceedings of the 2002 international conference on Compilers, Architecture, and Aynthesis for Embedded Systems*, pages 238–246, New York, NY, USA, 2002. ACM.

[33] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *SIGOPS Oper. Syst. Rev.*, 38(5):248–259, 2004.

[34] Y. Zhu and F. Mueller. Feedback EDF scheduling of real-time tasks exploiting dynamic voltage scaling. *Real-Time Syst.*, 31(1-3):33–63, 2005.