UNIVERSIDADE FEDERAL FLUMINENSE

VINICIUS TAVARES PETRUCCI

Optimization of power and performance for heterogeneous server systems

> NITERÓI 2012

UNIVERSIDADE FEDERAL FLUMINENSE

VINICIUS TAVARES PETRUCCI

Optimization of power and performance for heterogeneous server systems

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Doctor of Science. Topic Area: Parallel and Distributed Systems.

Advisor: Orlando Loques

> NITERÓI 2012

Optimization of power and performance for heterogeneous multiprocessing server systems

Vinicius Tavares Petrucci

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Doctor of Science.

Approved by:

Prof. Orlando Gomes Loques Filho, IC-UFF (President)

Prof. Julius Cesar Barreto Leite, IC-UFF

Prof. Eugene Francis Vinod Rebello, IC-UFF

Prof. Claudio Luis de Amorim, COPPE/PESC/UFRJ

Prof. Renato Fontoura de Gusmão Cerqueira, IBM-Research

Niterói, 20 December, 2012

"The journey is the reward"

Chinese Proverb.

Para Bete e Claudio.

Agradecimentos

- Um agradecimento mais que especial aos meus pais Claudio e Bete por todo amor, carinho, e apoio incondicional durante todos esses meus 29 anos. Sem eles, seria impossivel chegar ate aqui, nao apenas biologicamente falando. :-)
- À minha querida irma Juliana, sempre presente e tendo um carinho enorme comigo, e Nilson Victor (Foguinho), um irmao de consideracao, por todo incentivo e torcida, e por trazerem ao mundo duas figurinhas maravilhosas: Caio e Laura.
- Aos meus avos paternos Bras (in Memoriam) e Ecila, e maternos: Custodio (in Memoriam) e Merces.
- Ao Prof. Orlando Loques pela orientacao, apoio, e incentivo constantes durante todo meu periodo na pos-graduacao (mestrado e doutorado) na UFF.
- Ao Prof. Daniel Mosse pela inestimavel colaboracao (na pratica, co-orientacao) neste trabalho e imenso apoio (tanto na parte de trabalho/tecnica quanto social/amizade) durante meu estagio de doutorado, na Universidade de Pittsburgh, EUA.
- Big hugs and thanks to: Michelle, Phil, Mike, Evie, Lory, Peter, Mina, Rakan, Iyad, John, Hannah. All of you made my stay in Pittsburgh a very pleasant and memorable one!
- Prof. Julius Leite, Eduardo Uchoa e Enrique Vinicio Carrera pelas frutiferas discussoes tecnicas e colaboracao.
- Quando vim pra Niteroi para cursar o mestrado na UFF, morei durante o primeiro ano na casa do meu tio-avo Andre e Ceicao, que me acolheram como um filho. Tenho uma enorme gratidao por isso, e aos primos Bruno, Leo, Camila e Felipe, pelo companheirismo. Grande Tio Andre', que algumas semanas antes da minha defesa meu deu um abraco e me desejou boa sorte. Apos 3 dias da minha defesa, veio a falecer abruptamente aos 62 anos, apos um acidente estupido em uma caminhada pela praia. Como engenheiro eletrico, me lembro de nossas sempre agradaveis conversas (a noite, quando ele voltava do trabalho, durante uma pausa e outra do jogo de paciencia no computador) e de quando eu falava da minha tese em economizar energia, ele sempre curioso e interessado em saber mais e aprender. Me lembro tambem,

eu devia ter uns 15 anos, do tio Andre mexendo num circuito eletronico na sala do seu antigo apartamento e me explicando sobre uma patente daquele circuito que ele tinha desenvolvido. Tal exemplo de curiosidade e paciencia (muito alem daquela da tela do computador) em realizar e entender as coisas da vida (com seu jeito simples) me inspirou bastante. Obrigado tio Andre, descanse em paz.

- Agradecimento tambem especial ao primo Guilherme Saad Terra pelo grande incentivo em cursar uma pos-graduacao.
- À toda familia de Minas e Campos. Em especial aos primos Diego Tavares, Renato Tavares, Lucas Tavares, Donato Tavares, Luciano Petrucci.
- Aos amigos Alessandro Copetti, Thibaut Lust, Juliano Kazienko e Douglas Mareli pelas boas cantareiras :-)
- Aos amigos Gil de Goes, Leonardo Costa, Pablo Saraiva, Gustavo Lima, Filippe Mota, Carolina Laert, pelos otimos momentos juntos.
- Aos amigos que tive o grande prazer em conhecer durante meu tempo na UFF: Anand, Hugo, Sergio, Romulo, Puca, Carlitos, Luciano Bertini. Higor, Renatha, Eyder, Luciana, Juliana, Stenio, Jacques, Giulio, Matheus, Gleiph, Gustavo Zanatta, Gustavo Alexandre, Diego Brandao, Erick Passos, Ney Paranagua, Thibaut Vidal.
- Ao suporte e secretaria da pos-graduacao, em especial Rafael Abreu, Carlos Eduardo, Teresa Cancela e Viviane Aceti, pela gentileza e cooperacao.
- Conselho Nacional de Pesquisa (CNPq) pelo apoio financeiro durante o doutorado.
- CAPES pelo apoio financeiro durante o estagio de doutorado sanduiche na Universidade de Pittsburgh.

Abstract

An increasing number of clusters of server machines with multi-core processors have been deployed in large-scale computing environments (data centers). The energy consumed to maintain these server systems up and running is a very important concern that requires major investigation of optimization techniques to improve the energy efficiency of their computing infrastructure. This thesis develops, implements, and evaluates optimization approaches to determine and apply energy-efficient assignments of tasks to processors in a server system. Since task workload may change over time, these approaches include a self-adaptive scheme to dynamically change task-to-processor assignment at runtime, leveraging optimized decisions driven by optimization models and heuristic techniques.

This thesis describes specialized management strategy and implementation for the following server scenarios. In the first case, we consider a virtualized server environment where multiple services can be hosted on a single physical server. We determine an optimized subset of servers that must be active (and respective CPU speeds) and a corresponding mapping of the services to physical servers. In the second scenario, we deal with a heterogeneous multi-core platform that includes types of cores having different power and performance characteristics, namely fast/high-performance and slow/power-efficient core types. We provide optimized thread assignment decisions by mapping workload threads to run on the core type that is best suited for them, taking advantage of runtime observation of compute-intensive vs IO/memory-intensive execution phases of the threads. We show energy savings and performance gains for a variety of workloads, while respecting task real-time performance needs in the server system.

Keywords: Energy-efficient systems. Optimization. Server Virtualization. Multi-core Processors.

Contents

1	Introduction						
	1.1	.1 Problem statement					
	1.2	Optim	ization framework	2			
	1.3	The ca	ase of virtualized server cluster	5			
	1.4	The ca	ase of heterogeneous multi-core systems	7			
	1.5	Thesis	contributions	9			
2	Opt	Optimized management of virtual heterogeneous servers					
	2.1	Server	cluster modeling \ldots	12			
		2.1.1	Performance model	12			
		2.1.2	Power consumption model \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots	14			
		2.1.3	Power/performance model validation	15			
	2.2	Optim	ization for cluster management	17			
		2.2.1	Optimization model	17			
		2.2.2	Optimization control strategy	19			
		2.2.3	Dynamic optimization support	22			
	2.3	mentation in a cluster testbed	23				
		2.3.1	Server architecture	23			
		2.3.2	Cluster QoS measurement and control	25			
		2.3.3	Application workloads	27			
		2.3.4	Load prediction	28			
		2.3.5	Power and performance gains	29			
	2.4	Scalability concerns					
	2.5	Related work					
	2.6	Summary					
3	Thread assignment optimization for heterogeneous multi-cores						
	3.1	1 Optimized thread assignment					
		3.1.1	Optimization problem	39			
		3.1.2	Thread performance prediction	41			
		3.1.3	Solution to the optimization problem	43			
	3.2	Simula	ation methodology	44			
		3.2.1	Thread execution behavior	44			

		3.2.2	Core and memory system performance	45			
		3.2.3	Simulator environment	45			
		3.2.4	Estimating thread execution time	46			
	3.3	Simula	ation results	47			
		3.3.1	Energy-delay product	49			
		3.3.2	Tardiness	49			
		3.3.3	Memory contention analysis	51			
		3.3.4	Scalability of optimization scheme	52			
	3.4	Linux	${\rm implementation} . \ . \ . \ . \ . \ . \ . \ . \ . \ .$	54			
		3.4.1	Emulated heterogeneous core system	54			
		3.4.2	Workload description and measurements	56			
		3.4.3	Performance gains and energy savings	57			
		3.4.4	Best and worst case analysis	58			
	3.5	Relate	ed work	60			
	3.6	Summ	ary	61			
4	Proportional share scheduling for heterogeneous multi-cores						
	4.1	Fairne	ess in dynamic thread assignment	63			
	4.2	Power	/performance for heterogeneous cores	64			
		4.2.1	Thread performance/bias characterization	65			
	4.3	Lucky	scheduling	68			
		4.3.1	Energy efficiency metric	68			
		4.3.2	Algorithm outline	69			
	4.4	Result	$_{\rm S}$	70			
		4.4.1	Workload description	71			
		4.4.2	Energy efficiency and performance gains	72			
	4.5	Relate	ed work	74			
	4.6	Summ	ary	74			
5	Con	clusion	and future directions	75			
Aj	ppend	lix A -	Publications	77			
Bi	bliog	ranhv		79			
		- aping		10			

Chapter 1

Introduction

1.1 Problem statement

An increasing number of clusters of server machines with multi-core processors have been deployed in large-scale computing environments (data centers) [43]. These server platforms allow for the development and implementation of many kinds of services supporting different applications, such as email and online banking, in a scalable and efficient manner. Also, the applications running on these server clusters usually have great processing and performance demands, incurring in enormous amount of energy use and costs, and indirectly contributing to increase CO_2 generation and environmental deterioration [74]. According to [21], it is common to have server clusters that are built on the order of thousands to tens of thousands of physical servers, whereas medium clusters are on the order of hundreds of servers.

The energy consumed for keeping today's server systems running became a very important concern, which in turn, requires major investigation of techniques to improve the energy efficiency of their computing infrastructure [11, 28, 84]. According to a study by the Uptime Institute and McKinsey and Company [74], server clusters in data centers contribute to 30% of the world's carbon-dioxide emissions and will surpass those of the airline industry by 2020. A supercomputer such as Japan's Fujitsu K (10.5-petaflop) currently draws 12.7 megawatts which is enough to power a middle-sized town [58]. Other data center studies, such as [4], indicate that costs associated with power and cooling could overtake hardware acquisition costs.

More recently, the *New York Times* has published a report [78] about the energy use by data centers. According to their study, data centers currently use about 2% of total electricity in the US. One of the reasons for this high energy consumption is that Internet-based companies are typically running their data center facilities at maximum capacity, with server utilization rates between 7% and 12%, wasting 90% or more of the grid electricity consumed/paid. In a study of Google servers it was reported that typical server utilization is between 10 and 50 percent of peak, with a CPU utilization that rarely surpasses 40 percent [5]. Other studies estimate Amazon's EC2 utilization at 7% to 26% [68]. Figure 1.1 shows the server utilization histogram of a real HP data center running Web and IT services. We can observe that the servers in the data center spend most of time (60%) under a quite low utilization (10%).



Figure 1.1: Histogram of server utilization [76]

This thesis advocates the adoption of cost-effective optimization solutions for the sustainable development and efficient utilization of computing resources. We provide an optimization framework for periodic monitoring, evaluation and control/management to assess the energy efficiency in server systems. A technical challenge is that indiscriminately providing high server utilization can lead to negative effects such as long latency of user-request response, shared resource contention and performance degradation. Some workload demands can vary significantly over time and resource utilization control is essential in order to have room to handle workload spikes. Thus, delivering combined power/energy-efficient and performance requirements in server systems is a very challenging and important concern.

1.2 Optimization framework

We propose an optimization framework that includes a mathematical formulation and a scheme to dynamically determine optimized task-to-processor assignment, since task execution phases and performance requirements may change over time. Following the concepts of self-adaptive (autonomic) software systems [18, 19, 53], our framework is designed to periodically performs the following loop: (1) monitor the important system run-time measures to characterize performance requirements (e.g., CPU, memory bandwidth) of tasks, (2) solve a new optimization instance of the optimization model, given the updated values of the inputs, (3) use the optimized solution to configure the server system, for example, by dynamically migrating tasks to different processors based on task runtime behavior and/or adjusting CPU speeds according to the workload. Also, we may consider consolidating tasks into as few processors as possible by (quickly) powering down idle processors and bringing them up when thread workloads increase.



Figure 1.2: Optimization framework for self-adaptive server system

A performance Monitor module gathers run-time task data from the server system. The monitored values are evaluated by the Estimator module to predict the performance behavior of tasks running on different heterogeneous processors. To accomplish that, a performance prediction model can be derived analytically based on empirical analysis of tasks executions on different processor types. This module is also used to filter and estimate future values from past observed values for the workload time series.

After the monitoring and prediction phases, the Optimization model is updated with new inputs according to the dynamic behavior observation of each task running in the system and a new instance of the optimization problem is constructed. The Optimizer module solves the updated optimization problem instance, yielding a new optimized solution.

Note that the Optimization model represents the problem definition in a formal way. To solve such an optimization model, we rely on an efficient solver to achieve a low execution overhead, which depends on the problem/instance size. Alternatively, rather than solving directly the full optimization model, we may consider applying incremental local search/modification in the current optimized solution aimed at improving a given thread-to-core assignment for the next scheduling interval.

The Reconfiguration Algorithm module is responsible for applying the changes in the server system, such as migrating tasks and adjusting processors speeds/states, transitioning the system to a new state given by the new optimized configuration. Choosing the periodicity of migration and reconfiguration of the system allows for providing energy and performance gains with low overhead.

To develop our optimization framework, we consider a target server cluster scenario (shown in Figure 1.3) consisting of a group of replicated physical servers. The server cluster presents a single view to the clients through a special entity termed *dispatcher/load*

balancer server, which distributes the incoming workload among the actual servers that process the requests (jobs). The dispatcher server is also responsible for monitoring and managing the current configuration of the servers. In this system, we aim to guarantee specified quality-of-service requirements, such as service throughput and response time. Also, to reduce costs, the set of servers should be configured to minimize the power/energy consumption.



Figure 1.3: Server cluster environment

The first case addressed in this thesis deals with *cluster-level optimization* (Section 1.3) given by (1) which servers must be active and their respective CPU frequencies and (2) a corresponding mapping of the services to physical servers. We consider a virtualized server environment, where multiple services can be hosted on a single physical server. In the second scenario, this thesis addresses the case of *server-level optimization* dealing with heterogeneous multi-core processors (Section 1.4). Previous research has shown that improved performance and energy-efficiency benefits can be achieved by adopting heterogeneous multi-core systems in contrast to traditional homogeneous systems [61]. Such a heterogeneous platform includes cores with same ISA (instruction set architecture) but very different power and performance characteristics.

This thesis proposes approaches for cluster and server level optimizations that are implemented and evaluated separately. Nevertheless, these approaches can work cooperatively since they encompass different monitoring/management levels and timing constraints. That is, a longer control period (seconds to minutes) can be used for cluster level adaptations such as server on/off and VM migration/replication, whereas a shorter control period (milliseconds to seconds) can be used for thread assignment/scheduling on each multi-core server. This would allow for optimizing energy efficiency for both heterogeneous virtualized servers and individual multi-core processors. It is out of the scope of this thesis to develop of a comprehensive approach that implements and evaluates these two optimization schemes when used in an integrated solution.

Next, we will present in more detail our specialized management strategy and implementation for each multiprocessing optimization scenario.

1.3 The case of virtualized server cluster

Many of today's server clusters rely on virtualization techniques to run different VM's (*Virtual Machines*) on a single physical server, allowing the hosting of multiple independent services. With server virtualization, as shown in Figure 1.4, each application runs as a virtual machine that includes its own operating system (e.g., Linux) and any additional software or library required to support the respective service execution, such as a Web or e-mail services. These cluster architectures are common in utility/cloud computing platforms [21, 40], such as Amazon EC2 and Google AppEngine.



Figure 1.4: Server virtualization architecture and consolidation view

Server virtualization has been widely adopted in data centers around the world for improving resource usage efficiency and service isolation. In particular, several VM technologies have been developed to support server virtualization, such as Xen and VMware [3, 37]. From a power and performance management point of view, the adoption of virtualization technologies has turned power-aware optimization on clusters into an interesting research topic. Virtualization provides a means for server consolidation through on-demand allocation and live migration. This helps increase server utilization and reduce the long term use of computer resources and their associated power/energy consumption.

Specifically, the ability to dynamically distribute server workloads in a virtualized server environment allows for turning off physical machines during periods of low activity,

and bringing them back up when the demand increases. Moreover, server on/off mechanisms can be combined with CPU DVFS (*Dynamic Voltage and Frequency Scaling*), which is a technique that consists of varying the frequency and voltage of the CPU at runtime according to processing needs, in order to provide even better power optimizations.

The DVFS technique allows for dynamically adjusting the performance states (P-States) at which the server can operate when the CPU is active, which consists of a pre-defined set of frequency and voltage combinations. A DVFS-capable processor has only discrete levels of operating points. As an example of DVFS power management capability, Figure 1.5 shows the P-states varying frequency and voltage, and the effect on power consumption for the Intel Pentium M 1.6 GHz processor [31]. In practical terms, the processing performance of a server system is proportional to its CPU frequency.



Figure 1.5: Example of CPU operating states and power consumption ([31])

In addition to the DVFS technique, we assume that a server can be turned on/off in the cluster and the power dissipated by the servers when they are completely switched off can be further minimized. For example, a server can be switched off by a remote halt/suspend command via network message. To resume a machine from the suspended state, the servers can make use of the *Wake-on-LAN* mechanism available in Ethernet cards, which allows a machine to be turned on remotely by a network message, but consuming an extra power to keep active the Ethernet card and power supply unit (approximately 5W) [9].

The optimization range related to power consumption and possible energy savings are greater when it is possible to turn off servers compared to using only DVFS. On the other hand, the overhead/latency to perform DVFS (order of microseconds) is much lower than turning servers on/off (order of seconds). An optimized power usage can be achieved with combined DVFS and on/off mechanisms. Notice that Figure 1.5 only includes the power consumption of the CPU. A server is composed of other components that also consume power, such as hard disk, RAM memory, motherboard; however, power management for these components are not addressed in this thesis.

Virtual server consolidation must be carefully employed, considering individual ser-

vice performance levels, in order to guarantee QoS (*Quality of Service*) related to their corresponding SLA (*Service Level Agreements*). This is not a simple task since large clusters include many heterogeneous machines and each machine could have different capacity and power consumption according to the number of CPU cores, their frequencies, their specific devices, and so forth. Additionally, the incoming workload of a server can significantly change over time. Our solution to this scenario is to efficiently manage the cluster resources leveraging server virtualization and CPU DVFS techniques.

1.4 The case of heterogeneous multi-core systems

There is an increasing adoption of multi-core processors in server systems for improved performance and power usage while running parallel tasks in multiple cores. Most of today's computer systems are largely based on homogeneous multi-core processors, where all cores are the same. Due to the desire to reduce energy consumption, computer architecture researchers and designers are beginning to move to heterogeneous multi-cores [20, 34]. This heterogeneous system assumes that many low power/performance cores ("small" cores) can be combined with a few higher-performance cores ("large" cores) in a given multi-core processor [59, 67], as illustrated in Figure 1.6. A heterogeneous multi-core system can deliver higher power efficiency, such as performance per watt, when compared to homogeneous systems running a mix of distinct kinds of applications [29, 61]. While power efficiency is important, the server system must also conform to finishing their tasks in a timely fashion [23].



Figure 1.6: Multi-core system evolution: from homogeneous to heterogeneous cores

In homogeneous multi-core systems, there is a limitation in that the frequency can be scaled individually but all cores uses the same voltage (associated with the maximum core frequency) since they are fed from the same off-chip voltage regulator/domain [46]. There are some research efforts (simulation) that attempt to address the limitation of existing voltage regulator implementations by proposing several designs for on-chip voltage regulators, possibly in the future enabling fast adjustment of per-core frequency and voltage (on the order of nanoseconds) [56].

Over time, DVFS is becoming less attractive in homogeneous multicore systems [66], since it is very expensive to fully implement DVFS for individual cores on the same chip and low-power processors are already operating at near-minimum supply voltages [48]. Previous research in the multi-core area has shown that migrating threads between cores with distinct fixed frequency/voltages has comparable energy gains to systems having percore DVFS mechanisms [83]. Their research work advocates for real heterogeneous core systems to be adopted as an alternative energy-efficient design to homogeneous multi-core systems.

Having different core types in a multi-core system opens up new challenges and possibilities for power/energy management, thread scheduling and load balancing. In this scenario we obtain improvements in energy efficiency and performance by assigning each thread to run on the core type (big or small) that is best suited for it. To provide such improvements, thread-to-core assignment decisions take advantage of runtime observation of compute-intensive (big core bias) vs memory-intensive (small core bias) execution phases of the threads [29, 61].

Recent research has shown that two types of cores (*big* high-performance vs *small* lowpower) are able to capture most of the power/performance benefits from core heterogeneity while running typical workloads [20, 34, 59]. Additional energy-savings may be achieved by maximizing the utilization of a set of active cores and quickly powering down idle cores, bringing them up when thread workloads increase.

There are also challenges to meeting limited memory bandwidth constraints when running multiple threads in the heterogeneous multi-core system. The intuition is that moving specific threads between different core types at runtime may affect the rate of requests to the memory subsystem, overwhelming (or underwhelming) the memory controller/bus and thus delaying (or speeding up) the thread execution. In particular, moving some threads to small cores at runtime may decrease the requests on the memory subsystem, lowering thread execution throughput.

Additionally, large cores can have bigger private caches that may reduce the number of memory requests and associated thread execution time. Clearly, the best thread assignment decision should match the computational and memory demands of the running threads with the capabilities of the cores and memory subsystem. In addition to satisfying the threads' computational demands, thread scheduling has to provide some degree of fairness to the allocation of the platform resources to the threads in the system. Otherwise, some threads can monopolize the available big cores and hinder the progress of other threads.

1.5 Thesis contributions

This thesis develops, implements, and evaluates optimization schemes to determine and apply energy-efficient assignments of tasks to processors in a server system. Our optimization framework allows for exploration of several domains. We consider **tasks** as (a) *virtual machines*, which run specialized operating system plus library/software applications, or (b) *threads/processes* that run on a typical operating system. We consider the resource allocation problem for two scenarios: a cluster of (i) *machines* linked by a fast network and (ii) processing *cores* in a physical package/chip.

The outline and main contributions of this thesis are as follows:

- Following the self-adaptation concepts outlined in Section 1.2, we pose the taskto-processor management as one of optimization problem, incorporating it in the design of a dynamic configuration framework. The basic idea is to solve a new instance of an optimization model, given the task runtime input values, and use the optimized solution to configure the server system.
- In Chapter 2, we address the case of virtualized servers by deriving a simple but effective way of modeling power consumption and capacity of servers for heterogeneous machines and changing workloads. Given this power/performance model, we develop an optimization model and strategy to determine power-efficient cluster configurations (*i.e.*, which servers must be active and their respective CPU frequencies). In this platform, the tasks/services are implemented as virtual machines hosted on several shared heterogeneous servers with task workloads varying over time.
- In Chapter 3, we propose an optimization model to determine the most suitable thread assignment decision, while meeting soft real-time requirements for the application threads and minimizing energy consumption in the heterogeneous multi-core system. This case follows the trend to move from homogeneous to heterogeneous multi-core systems to provide better performance and energy-efficiency benefits.
- Finally, in Chapter 4, we design and implement a heuristic for energy-efficient thread assignment, *lucky*, which is based on lottery scheduling to provide fair-share of available heterogeneous cores in the multi-core system. We argue that thread scheduling should provide fair allocation of the platform resources to the threads, in addition to satisfying the threads' computational demands. We propose a simple and effective way to determine ticket assignment for thread workloads by estimating thread performance and energy efficiency between core types in the system.

In addition to the research in Chapters 2-4, we add in each chapter a section for related work. Lastly, we summarize this dissertation and discuss future research directions in Chapter 5.

Chapter 2

Optimized management of virtual heterogeneous servers

Following the optimization framework introduced in Chapter 1, our scheme periodically solves an optimization model and uses the optimized solution to configure the cluster. In this way, considering that the tasks have individual time-varying workloads, our optimization strategy enables the virtualized server system to react to load variations and adapt its configuration accordingly. The proposed optimization strategy also accounts for the overhead due to switching servers on/off and disruptive migration of virtual servers. Furthermore, the optimization determines an optimized load distribution for the applications, which is a non-trivial task considering that an application workload can be distributed among multiple heterogeneous servers in a cluster. The underlying mathematical formulation for optimizing the power and performance in a virtualized cluster is given by a mixed integer programming (MIP) model.

We describe our approach to virtualized server scenario in two parts. First, we provide a simple and effective way of modeling performance and power consumption of the servers in the cluster (Section 2.1). Second, based on the power and performance models, we describe an optimization formulation implemented in a control loop strategy for power and performance management in a virtualized server cluster (Section 2.2). The system model and server architecture testbed used to apply our optimization approach are described in Section 2.3. In Section 2.3.5, we evaluate our approach through experiments driven by actual workload traces. We present in Section 2.4 simulation results and discussion regarding scalability issues.



Figure 2.1: Server performance at maximum CPU frequency as a function of the normalized CPU utilization

2.1 Server cluster modeling

2.1.1 Performance model

In order to model the performance of each real machine, the following definitions are introduced:

- M is the number of applications or services running on the cluster.
- N is the number of physical servers in the cluster.
- f_{ij} is a valid working frequency $j \in \{1 \dots F_i\}$ for the CPU on a given server $i \in \{1 \dots N\}$.
- u_{ij} is the normalized utilization of the CPU at server-frequency f_{ij} .
- $r_{ij}(u_{ij})$ is the maximum number of requests of certain type that a server *i* can attend per unit of time when running at frequency *j* and its CPU utilization is u_{ij} .
- R_{ij} is the maximum r_{ij} (*i.e.*, when $u_{ij} = 100\%$).

Figure 2.1 shows the linear relationship existing between the number of completed requests per second and the normalized CPU utilization for five different machines in our server cluster (described in more detail in Section 2.3). The web requests used in the experiments are CPU-bound and consume a fixed amount of CPU time, and the frequency of the CPU was kept constant. Specifically, we measured the performance of the servers, for each CPU speed, in terms of the number of requests per second (req/s) that they can handle at a given target CPU utilization. To generate the benchmark workload, we



Figure 2.2: Server performance while varying CPU frequency

used the *httperf* tool [77]. Note that the case of web servers may be similar to other CPU-intensive services in a cluster.

We consider the CPU resource as the bottleneck in our system. The required data for the web services are already cached in main memory after the system has been running. Since the performance of a server is proportional to its CPU utilization at constant frequency, the server performance can be modeled as:

$$r_{ij}(u_{ij}) = R_{ij} \cdot u_{ij}$$

The performance of a server is also proportional to its CPU frequency. Figure 2.2 shows the relationship existing between these two variables. Since this relationship is also linear, the server performance can be expressed as:

$$r_{ij}(u_{ij}) = r_{iF_i}(u_{ij}) \cdot \frac{f_{ij}}{f_{iF_i}}$$

Based on the two previous relationships, we can conclude that the performance of a machine can be modeled as:

$$r_{ij}(u_{ij}) = R_{iF_i} \cdot u_{ij} \cdot \frac{f_{ij}}{f_{iF_i}}$$

where R_{iF_i} and f_{iF_i} are the only constants defined for each server. Thus, we define the capacity of a server-frequency by

$$c_{ij} = R_{iF_i} \cdot (f_{ij}/f_{iF_i})$$

In addition, the normalized utilizations of the CPU at different frequencies when the workload is kept constant can be related by: $u_{ik} = u_{ij} \cdot \frac{f_{ik}}{f_{ij}}$ where k and j are valid CPU frequency levels in a given server i.

Since we consider a shared server cluster with multiple distinct applications, it is quite impractical to assume *a priori* any information about the incoming workload for those applications because they all have different types of requests with diverse processing requirements. Thus, we define and measure the workload demand of an application by considering the utilization of the bottleneck resource in the system, which is the CPU in our case. More formally, for each application $k \in \{1 \dots M\}$, we define

$$d_k = \sum_{i=1}^N u'_{ik} \cdot R_{iF_i}$$

to represent the workload of application k basically in terms of the sum of the CPU usage, given by the variable u'_{ik} , which is monitored from the running system, for each server i allocated to that application k.

2.1.2 Power consumption model

To model the power consumption of each real machine, besides the previous terminology, some additional terms are required:

- $p_{ij}(u_{ij})$ is the average power consumption of a server-frequency f_{ij} when its CPU utilization is u_{ij} .
- PM_{ij} is the maximum (busy) power consumption of a server-frequency when $u_{ij} = 100\%$ at f_{ij} .
- Pm_{ij} is the minimum (idle) power consumption of a server-frequency when idle at f_{ij} .

Figure 2.3 shows the measured relationship existing between the power consumed by a given server and its performance at 2.57Ghz. We obtained a similar relationship for the other available frequencies. Since the relationship between these two variables is linear when the CPU frequency is kept constant, the power consumed by a server can be modeled as:

$$p_{ij}(u_{ij}) = Pm_{ij} + (PM_{ij} - Pm_{ij}) \cdot u_{ij}$$

To obtain the several PM_{ij} and Pm_{ij} values, we can use the relationship displayed in Figure 2.4. This shows that the power consumed by a server is proportional (in a quadratic form) to its CPU frequency. Thus, the server power consumption can be modeled as:

$$PM_{ij} = PM_{i1} + K_M \cdot (f_{ij} - f_{i1})^2$$



Figure 2.3: Server power consumption as a function of the server performance keeping constant the CPU frequency

$$Pm_{ij} = Pm_{i1} + K_m \cdot (f_{ij} - f_{i1})^2$$

where $K_M = (PM_{iF_i} - PM_{i1})/(f_{iF_i} - f_{i1})^2$ and $K_m = (Pm_{iF_i} - Pm_{i1})/(f_{iF_i} - f_{i1})^2$. In other words, the power consumption of a server basically depends on its working frequency (f_{ij}) , its normalized CPU utilization (u_{ij}) , and the constants PM_{i1} , Pm_{i1} , PM_{iF_i} and Pm_{iF_i} .

Although related works, such as [33], consider a cubic relationship between power consumption and the frequency at which a server is running, especially for the processor, we are assuming a quadratic relationship between these two variables. Because our main goal is to predict power consumption values based on only four power measurements (*i.e.*, Pm_{i1} , Pm_{iF} , PM_{i1} , and PM_{iF} for a given server *i*), we have proposed a more straightforward alternative, given by the previous power equations, to the curve fitting processes. Using this simplified power model, we observed that the lowest mean squared errors are obtained with the quadratic relationship instead of the cubic one. As we can see in Section 2.1.3, the normalized root mean squared errors for the power vs. frequency relationship are always less than 2%. When we used the cubic relationship, the normalized root mean squared errors were as big as 5%.

2.1.3 Power/performance model validation

To validate our analytical model, we have measured the performance and power consumption of five different machines in our cluster, varying their CPU frequency and CPU utilization. The machines used on our experiments, termed Edison (Intel Core i7), Farad (Intel Core 2 Duo), Galvani (Intel Core i5), Gauss (AMD Athlon 64) and Tesla (AMD Phenom X4), have very different technologies including a wide range of working frequen-



Figure 2.4: Server power consumption as a function of the CPU frequency at 100% utilization

cies, whose main characteristics are described in Section 4.2.

In our case, recording the average values for power and performance for a given load over a period of 2 minutes was sufficient to obtain a good average and low standard deviation. We measured the average power consumption and performance for different levels of load in 10% increments for each server at each CPU frequency available. We measured AC power directly using the *WattsUP Pro* power meter with 1% accuracy [25]. The power measurements thus represent the whole machines, not only their CPUs.

In our experiments, due to incompatibility issues on the Linux kernel with the Intel "Turbo Boost" feature, we disabled it on Edison and Galvani machines. Specifically, the processors can achieve different speeds than those requested by the Linux kernel power management when the "Turbo Boost" was enabled. In addition, the "Turbo Boost" feature makes it quite difficult to sample the exact maximum CPU speed and performance of the processor, because these values would depend on several factors, such as the chip temperature, which requires further study on this technology.

Table 2.1 summarizes our measurements. Comparing the measured values of performance with our model, the maximum normalized root mean squared error for the performance–CPU utilization relationship (r vs u) is 2.5%. For the performance–CPU frequency relationship (r vs f), the normalized root mean squared error is less than 1.7% in all the cases. Similarly, the comparison of the measured values of power with our model shows that the power–CPU utilization relationship (p vs u) has a maximum normalized root mean squared error of 1.5%. The normalized root mean squared error for the power–CPU frequency relationship (p vs f) is always less than 1.7%.

Machine	$\mathbf{r} \ vs \ \mathbf{u}$	$\mathbf{r} vs \mathbf{f}$	$\mathbf{p} vs \mathbf{u}$	$\mathbf{p} \ vs \mathbf{f}$
Farad	1.5%	0.8%	0.4%	0.0%
Galvani	1.8%	0.8%	1.5%	1.7%
Edison	1.5%	0.8%	0.3%	1.2%
Tesla	1.7%	1.7%	0.2%	1.2%
Gauss	2.5%	1.3%	0.8%	1.4%

Table 2.1: Normalized root mean squared errors

Although the normalized root mean squared errors are quite small, it is also important to mention that the maximum absolute error is always less than 8.3% (Tesla in the performance–CPU utilization relationship), which validates our analytical model with acceptable accuracy. Given these accurate power and performance relationships, we simplify the extensive and time-consuming task of power and performance benchmark, which commonly leads to high customization and setup costs for archiving power-aware optimizations in server clusters [41, 85]. In our simplified benchmark, we only needed to measure, for each machine, the idle and busy power at minimum and maximum CPU speed. Thus, in our cluster, it took four measurements per machine.

2.2 Optimization for cluster management

The cluster optimization problem we consider is to determine the most power efficient configuration (*i.e.*, which servers must be active and their respective CPU frequencies) that can handle a certain set of application workloads. The problem of allocating a set of processes across a pool of server machines is basically a bin-packing problem where each of the machines is a bin. The goal is to allocate as many processes as possible into each bin. The underlying mathematical formulation for minimizing the power consumed while meeting performance requirements in the virtualized cluster problem is given by a MIP (*Mixed Integer Programming*) model. The cluster configuration problem is NP-hard, since it can be seen as a generalization of the 1-D Variable-sized Bin Packing Problem, which is known to be NP-hard [32], when each type of server has an unlimited number and only a single frequency available.

2.2.1 Optimization model

The information about power consumption and capacity of servers can then be used by our optimization model to make runtime decisions. In our virtualized environment, the applications are implemented as VMs, which are assigned to physical servers. In our optimization model, several different VMs can be mapped (consolidated) to a single server. This way, our strategy for power-aware optimization in a cluster of virtualized servers is to select the physical servers where the VMs will be running and the frequencies of each real machine according to the CPU utilization that every VM associated to an application requires. Our optimization model also allows an application to be implemented using multiple VMs, which are mapped and distributed to different physical servers. This is useful for load balancing purposes, that is, when an application workload demands more capacity than the supported by a single physical server.

In addition to the terminology introduced previously, the following decision variables are defined: x_{ijk} is a binary variable that denotes whether server *i* uses frequency *j* to run application k ($x_{ijk} = 1$), or not ($x_{ijk} = 0$); y_{ij} is a binary variable that denotes whether server *i* is active at frequency *j* ($y_{ij} = 1$), or not ($y_{ij} = 0$). The u_{ij} variable denotes the utilization of server *i* running at frequency *j*.

The problem formulation for the cluster optimization is thus given by the following MIP model:

$$\begin{aligned} \text{Minimize} \quad \sum_{i=1}^{N} \sum_{j=1}^{F_i} [PM_{ij} \cdot u_{ij} + Pm_{ij} \cdot (y_{ij} - u_{ij})] \\ + swt_cost(U, y) \\ + mig_cost(A, z) \end{aligned} \tag{2.1}$$

Subject to

$$\sum_{k=1}^{M} d_k \cdot x_{ijk} \le c_{ij} \cdot u_{ij} \qquad \forall i \in \{1 \dots N\}, \forall j \in \{1 \dots F_i\}$$

$$(2.2)$$

$$\sum_{i=1}^{N} \sum_{j=1}^{F_i} x_{ijk} = 1 \qquad \forall k \in \{1 \dots M\}$$
(2.3)

$$\sum_{j=1}^{F_i} y_{ij} \le 1 \qquad \qquad \forall i \in \{1 \dots N\}$$

$$(2.4)$$

$$u_{ij} \le y_{ij} \qquad \forall i \in \{1 \dots N\}, \forall j \in \{1 \dots F_i\}$$

$$(2.5)$$

$$x_{ijk} \in \{0,1\}, \ y_{ij} \in \{0,1\}, \ u_{ij} \in [0,1]$$
 (2.6)

The objective function given by Equation (2.1) is to find a cluster configuration that minimizes the overall server cluster cost in terms of power consumption. The power consumption for the servers is given by the analytical model from Section 2.1. The objective function has also two terms to account for server switching and VM relocation costs. To model a server switching cost, we have included in the model a new parameter input $U_{ij} \in \{0, 1\}$ to denote the previous cluster usage in terms of which machines are turned on and off; that is, $U_{ij} = 1$ if machine *i* is running at speed *j*. Similarly, the new parameter input $A_{ik} \in \{0, 1\}$ denotes which application was previously associated with which server. More precisely, we may define the switching cost function as follows: $swt_cost(U, y) = \sum_{i=1}^{N} \sum_{j=1}^{F_i} [SWT_P \cdot (y_{ij} \cdot (1 - U_{ij}) + U_{ij} \cdot (1 - y_{ij}))]$. The constant SWT_P represents a penalty for turning a machine off (if it was on) and for turning a machine on (if it was off), which can mean an additional power consumed to boot the respective server machine. We do not consider the cost of changing frequencies, since the overhead incurred is quite low [85]. Actually, if $U_{ij} = 1$ for a given server $i \in \{1 \dots N\}$, we set $U_{ij} = 1$ for all $j \in \{1 \dots F_i\}$ to avoid taking into account frequency switching costs.

To facilitate modeling the VM relocation cost, we define a new decision variable $z_{ik} \in \{0, 1\}$ to denote if an application k is to be allocated on server i. We also include in the model a new set of constraints, which are defined as follows: $x_{ijk} \leq z_{ik} \quad \forall i \in \{1 \dots N\}, \forall j \in \{1 \dots F_i\}, \forall k \in \{1 \dots M\}$. The relocation cost function can be defined similarly based on the previous allocation input variable A and the new allocation decision z, such as $mig_cost(A, z) = \sum_{i=1}^{N} \sum_{k=1}^{M} [MIG_P \cdot (z_{ik} \cdot (1 - A_{ik}) + A_{ik} \cdot (1 - z_{ik}))]$. We assume that both server switching on/off and relocation penalties can be estimated in a real server cluster. In our case, these costs were basically defined by the average power consumption of the servers in the cluster.

In the optimization model, the constraints (2.2) prevent a possible solution in which the demand of all applications $k \in M$ running on a server *i* at frequency *j* exceeds the capacity of that server, given by our analytical model (Section 2.1). The constraints (2.3) guarantee that a server *i* at frequency *j* is assigned to a given application *k*. The constraints (2.4) are defined so that only one frequency *j* on a given server *i* can be chosen. The constraints (2.5) are used to relate the decision variable y_{ij} with the u_{ij} variable in the objective function. The solution is thus given by the decision variable x_{ijk} , where *i* is a server reference, *j* is the server's status (i.e., its operating frequency or inactive status, when j = 0), and *k* represents the respective allocated application. The expressions on (2.6) define the variables of the problem.

For example, a MIP solution that returns $x_{123} = 0.3$ and $x_{273} = 0.7$ means that 30% of the workload of application 3 is executed on server 1 running on the CPU frequency 2, and 70% of that workload is executed on server 2 running at frequency 7. The optimized solution also provides useful information to implement a load balancing scheme, such as a weighted round-robin policy, for distributing the application workloads among the servers in the cluster.

2.2.2 Optimization control strategy

Dynamic optimization behavior is attained by periodically monitoring the proper inputs of the optimization model and solving a new optimal configuration given the updated values of the inputs. In other words, as some of the input variables may change over time, such as the application workload vector, a new instance of the optimization problem is constructed and solved at run-time. We assume that the workload does not change radically often and it is mostly stable during the specified control period.

Particularly, we are able to devise a control loop of the following form: (a) monitor

and store the most recent values of the optimization input variables, (b) construct and solve a new optimization problem instance, yielding a new optimal configuration and (c) apply the changes in the system, transitioning the system to a new state given by the new optimized configuration. The details are given in the following algorithm.

```
do periodically:
```

```
// 1. Input variables
curDemand = getDemandVector()
curUsage = getCurrentUsage()
curAlloc = getCurrentAlloc()
// 2. Workload filter / prediction
d = predict(curDemand)
// 3. Run optimization
newUsage, newAlloc = bestConfig(d)
// 4. Generate usage and alloc sets for changes
chgUsage = sort(diff(newUsage, curUsage))
chgAlloc = sort(diff(newAlloc, curAlloc))
// 5. Power management operations
for (i, j) in chgUsage:
    if j == 0:
        turnOff(i)
    else:
        if curUsage[i] == 0:
            turnOn(i)
        setFreq(i, j)
// 6. Virtualization management operations
for (k, i) in chgAlloc:
    if i == 0:
        stopVm(k, curAlloc[k])
    else:
        if curAlloc[k] == 0:
            startVm(k, i)
        else:
            migrateVm(k, curAlloc[k], i)
```

The control loop outlined above relies on the mathematical formulation described in

Section 2.2.1 to solve the cluster configuration problem. The key idea of the optimization control policy is to periodically select and enforce the lowest power consumption configuration that maintains the cluster within the desired performance level, given the time-varying incoming workload of multiple applications.

The input variables for the control loop algorithm, described in step 1, are: the monitored and updated application demand (load) vector, the current server configuration and application allocation. At step 2, we may apply a predictive filter to estimate the workload demand vector for lookahead horizons. The **bestConfig** operator, at algorithm step 3, returns a cluster usage and allocation solution, where **newUsage** represents an usage configuration of servers and their respective status (i.e., its operating frequency or inactive), and **newAlloc** represents which application has to be associated with each server.

In fact, the configuration to be imposed is a difference between two sets: the new configuration and the current configuration solution. For example, suppose the current cluster usage is curUsage={(1,0),(2,2),(3,0)} and the new usage is newUsage={(1,0),(2,0), (3,4)}. Thus, we need to perform a change in the system given by chgUsage = newUsage - curUsage = {(2,0),(3,4)}. That is, we need to turn off server 2, and turn on server 3 at the frequency 4. To handle this, we apply a diff operator in the usage and allocation solutions provided by the optimization operator (see the algorithm step 4).

The order in which the operations are executed may lead to a problematic behavior. Specifically, in the example above, if the new usage configuration shutdowns the current active server before the new server is ready to respond the requests, as the server booting time is not instantaneous, the cluster will be in an unavailable state. To solve this issue, we simply sort the new cluster usage representation so that the operation to shutdown servers is always performed last, and the operations to increase frequency and turn on servers, respectively, are performed first. This scheme can be similarly adopted to the new allocation representation, in which the operations to start and migrate virtual machines are performed at first. To achieve this, we make use of a **sort** operator in the configuration algorithm, as shown in algorithm step 4.

At step 5, we employ dynamic configurations for power optimization which consists of (a) turning servers off in periods of low activity and turning them back on when the demand increases, and (b) exploiting the dynamic voltage and frequency scaling capability of current processor architectures to further reduce power consumption. Finally, to manage the application services (which are associated to virtual machines), we rely on configuration operators to start, stop, and migrate the virtual machines in the server cluster, such as those described in the algorithm step 6.

Execution example. As an example of optimization execution, we may assume that the demand vector (in request/sec) for three different applications is d=[45,120,17].

After solving the optimization problem, we have an abstract configuration solution given by a vector of tuples (i, j, k), where *i* is a server, *j* is the respective CPU speed, and *k* is the allocated application, defined by **conf** = [(1,3,1), (1,3,2), (1,3,3)]. This means that, application 1, 2 and 3 are hosted by server 1 at frequency 3, which is its maximum frequency, while the other servers are turned off.

At another execution snapshot, say d=[45,170,17], the new configuration solution is defined by conf = [(2,1,1),(1,3,2),(1,3,3)]. This means that if the demand for application 2 has been increased from 120 to 170, we need to turn on a new server and migrate application 1 to this new server 2 that will run at frequency 1 (i.e., the minimum frequency). In case the demand for application 2 decreases, we may turn off server 2 to save power and migrate back application 1 to server 1.

2.2.3 Dynamic optimization support

In our approach, the monitoring and dynamic configuration mechanisms (for example, as used by our optimization policy in Section 2.2.2) are implemented in terms of an application programming interface (API) [18, 19] given by the system support level (see Figure 2.5). For example, the Apache web server [94] supports an API to enable developers to extend a server with their own extension modules. The Xen hypervisor [3] also provides capabilities and mechanisms to monitor and manage virtual machines in a server cluster by means of an API.



Figure 2.5: API support for dynamic optimization

The key idea of an API is that it specifies an abstract and well-defined interface to control the behavior of the system, which builds on lower-level mechanisms at the system level. A desired feature for an API is that it can be called from several programming languages and is available as a remote procedure call, such as the XML-RPC protocol. Our optimization approach encapsulates most of the required functionality for dynamic configuration in terms of an API and provide generic configuration operators [80, 81]. This, in turn, enables the dynamic optimization logic to be described in a more appropriate way by using a number of high-level constructs. Examples of our API and operators include a call to a configuration operator, termed **bestConfig**, which encapsulates an optimization algorithm for solving the cluster configuration problem.

2.3 Implementation in a cluster testbed

In our virtualized cluster environment, targeted at hosting multiple independent web applications, we need to maintain two correlated objectives. First, we aim to guarantee quality-of-service requirements for the applications, for example, by controlling average cluster load. Second, to reduce costs, the set of currently active servers and their respective processor's speeds should be dynamically configured to minimize the power consumption.

2.3.1 Server architecture

The target testbed architecture (shown in Figure 2.6) consists of a cluster of servers running CentOS Linux 5.5. The cluster presents a single view to the clients through the front-end machine, which distributes incoming requests among the actual servers that process the requests (also known as workers). The worker servers run Xen 3.1 hypervisor enabled to support the execution of virtual machines and capable of CPU DVFS. Our testbed includes a cluster of seven physical machines, whose characteristics and configurations are detailed in Table 2.2.

The web requests from the clients are redirected, based on a load balancing scheme, to the VMs allocated to the applications that run Apache web servers on top of physical server machines. Each VM has a copy of a simple CPU-bound PHP script to characterize a web application. We define three different applications in the cluster, named App1, App2 and App3. To generate the workload for each application, we developed a workload generator tool to simulate realistic workload patterns (described in Section 2.3.3).

The load generator machine is physically connected to the front-end machine via a gigabit ethernet switch. The worker machines and the front-end machine are connected via another gigabit switch. The front-end machine has two gigabit network interfaces, each one connected to one of the switches. All worker machines share an NFS (Network File System) storage mounted in the front-end to store the VM images and configuration files.

Notice that the time to switch on a server, when a server was switched off by a halt or shutdown command, is on average 1.5 minutes. The shutdown time is about 15 seconds and halt power consumption is 6.5 W on average. In the future experiments, we believe that this time could be minimized by using mechanisms of suspend/hibernate on the servers. However, the suspend/hibernate mechanism is not supported yet by the Xen-enabled kernel in our servers.

The front-end machine is the main component in the architecture including three



Figure 2.6: Testbed server architecture

Machine	Role	Processor model	Cores	CPU model	RAM	Freq. steps
Xingo	Load generator	Intel Core i7	4	2.93	8GB	
Henry	Front-end	AMD Athon 64	1	2.2GHz	4GB	
Edison	Worker	Intel Core i7	4	2.67GHz	8GB	9
Farad	Worker	Intel Core 2 Duo	2	$3.0 \mathrm{GHz}$	6GB	2
Galvani	Worker	Intel Core i5	4	$2.67 \mathrm{GHz}$	8GB	12
Gauss	Worker	AMD Phenom X4 II	4	$3.2 \mathrm{GHz}$	8GB	4
Tesla	Worker	AMD Athon 64 X2	2	3.0GHz	6GB	8

Table 2.2: Specification of the machines used in our cluster

entities: (a) VM manager, (b) Load balancer, and (c) Controller. The VM Manager is implemented using the OpenNebula toolkit [79] which enables the management of the VMs in the cluster, such as deployment and monitoring. Since we are running virtualized web servers in our cluster, the Load Balancer implements a weighted round-robin scheduler strategy provided by the Apache's mod_proxy_balancer module [94]. Finally, the Controller implements the control loop strategy that consists of an external module written in Python that relies on the primitives provided by the VM Manager and Load Balancer modules. The goal of the Controller is to dynamically configure the applications over the server cluster, in order to reduce power consumption, while meeting the application's performance requirements. The Controller: (a) monitors application's load, (b) decides about a possible relocation in the cluster to attend the demand, based on the proposed optimization model, and (c) applies the new configuration executing necessary primitives (see details of the optimization strategy in Section 2.2).

For the integration of the Controller with the VM Manager module, we used the XML-RPC interface provided by OpenNebula. To dynamically manage the Load Balancer module, we implemented and installed in the front-end machine a new Apache module

written in C called *mod_frontend*, which relies on the Apache proxy load balancer module functionalities. This Apache module exposes a generic interface (through the XML-RPC protocol) to enable (and disable) a worker server, to assign weights in the load balancing scheme to the servers, and to monitor the cluster QoS properties, such as arrival rate, throughput and response time of the web requests execution. To manage the cluster machines, we issue commands remotely via SSH, for example, to turn servers off and to adjust servers frequencies. To resume a machine from the shutdown state, the servers support the *Wake-on-LAN* mechanism, which allows a machine to be turned on remotely by a network message.

2.3.2 Cluster QoS measurement and control

In this work, the cluster system load is measured by the sum of CPU usage of the virtual machines (running the applications) in the cluster. Because we are interested in the macro behavior of our optimization, the web applications are simplified in that we assume that there is no state information to be maintained for PHP requests. As for performance optimization guarantees, while still meeting temporal QoS requirements of the requests in the cluster, we provide a tight control on the system load; that is, the CPU utilization in the whole cluster. With such a control strategy, we show that we can also satisfy the user-perceived experience commonly measured by the response time of the requests in the cluster.

As shown in Figure 2.7, when the CPU utilization of a VM (running a web server application) is low, the average response time is also low. This is expected since no time is spent queuing due to the presence of many concurrent requests. On the other hand, when the utilization is high, the response time goes up abruptly as the server utilization gets close to 100%. In the graph, the maximum value for utilization is 400% because we are using quad-core machines and each core represents 100%. To meet response time requirements, we shall perform dynamic optimizations in the cluster before the machine reaches saturation, for example, above a target of 80% (or 300% for the quad-core configuration) of server utilization. This gives us a simple but effective measure to keep the response time under control. Moreover, since we measured the bottleneck resource (CPU) as a "black-box", this control scheme can work for different types of HTTP requests with distinct average execution times. This also allows for an extra amount of CPU capacity available to be used by the VM management domain (Dom0) on the physical servers during the migration or replication activities.

The response time is defined as the duration between the time a response is generated and the moment the server accepts the request. To obtain the response time for the web applications we have implemented a new Apache module (as mentioned in Section 2.3.1) that collects such timing information using pre-defined hooks provided by the Apache



Figure 2.7: Relationship among throughput, response time, and CPU utilization

Module API [94]. The hooks used to measure the response time are: post_read_request and log_transaction. The post_read_request function allows our module to store the moment a request was accepted by Apache and the log_transaction function allows it to store the moment a response was sent to the client.

Based on these hooks, we can also measure the workload *arrival rate* by accumulating the number of requests for a given interval in the post_read_request phase. Similarly, we can measure the *throughput* in the log_transaction phase. To smooth out high short-term fluctuations in these measurements readings, we have implemented a filter procedure in our Apache module based on a single exponential moving average. In the filter module, we have used $\alpha = 0.6$ as the default smoothing factor. Based on our experiments, this value was found suitable.

We mainly define the QoS for the applications in the server cluster by means of the maximum allowed response time for the respective requests — the application deadline. In the optimization approach, we should satisfy this QoS metric by managing two variables. First, the cluster utilization must be below a reference value, in our case $target_util = 80\%$. Second, in case the current response time is *above* the deadline response time, we determine a tardiness value by the ratio of current response time to the deadline response time, which denotes "how far" the average application requests are from the respective requests deadline.

More precisely, to guarantee that the response time restriction is met, we regulate the application workload demand vector by multiplying it to the *tardiness* preset limit to achieve the desired response time requirement. In a similar manner, to maintain the cluster utilization around its reference value, we normalize the application workload demand vector by the *target_util* to achieve the desired cluster utilization bound. Typically these reference values for utilization and requests deadline depends on workload demand
characteristics and performance purposes for a target cluster system.

That way, the target workload demand for a given application k, which was previously defined in Section 2.1.1, can be rewritten as $d'_k = (d_k \ target_util) * tardiness * \gamma$. A reinforcement constant γ can be used to give smoothing or boosting performance effect to allow the cluster to be more or less responsive to changes with respect to the response time tardiness metric. In our case, we assume a $\gamma = 1.0$, but other values for it need to be investigated in future experiments. So, the modified workload demand aims to drive our controller to dynamically configure the cluster to meet the QoS goal accordingly.

2.3.3 Application workloads

We generated three distinct workload traces using the 1998 World Cup Web logs to characterize the multiple applications in the cluster [2]. We calculated the number of requests over a fixed and customized sampling interval for different parts of the World Cup logs over time. We also adapted the original workload data to fit our cluster capacity, which is measured in requests per second, for a fixed type of request. As shown in Figure 2.8, the applications within the cluster can have a wide range of request demands. Each application in the cluster is assigned a maximum allowed response time specified at 500ms. In a real system, the deadline response times are given based on particular requirements of the applications and can assume different target values, which is allowed by our approach.



Figure 2.8: Workload traces for three different applications

Each workload demand point in Figure 2.8 is the average of 36 seconds, describing an experiment execution of 2 hours in duration. In our experiments, we assume that App1, App2, and App3 services can be distributed and balanced among all servers in the cluster. To generate the workloads, we implemented a closed-loop workload generator written in C that sends web requests to the applications. The idea employed was to dynamically change the "think time" of a given set of running web sessions (implemented using P-threads) within the system every 36 seconds, which is the time granularity of our workload traces. Specifically, the workload generator interprets each data point in Figure 2.8 as the target load to generate, deriving a new adjustment on the "think time" for the running web sessions, during each 36-second interval. Clearly, the "think time" is inversely proportional to the workload intensity; that is, the smaller the value of the "think time", the greater the workload intensity.

2.3.4 Load prediction

Previous works show that CPU server utilization [24] and Web server traffic [7, 88] can be modeled through simple linear models. Dinda and O'Hallaron [24] conclude that the host load is consistently predictable by practical models such as AR, reaching a very good prediction confidence for up to 30 seconds into the future. On the other hand, Baryshnikov *et. al.* [7] prove that even bursts of traffic can be predicted far enough in advance using a simple linear-fit extrapolation. Based on that, our predictor module implements a linear regression based on k past values to predict the load of the applications in the cluster.

To anticipate fast load increments, we must take into account the time for turning servers on. In order to avoid on/off disruptions we must consider the *break-even threshold* of the servers. That is, the time required by an unloaded server to consume as much energy as required for turning a server off and on immediately. In other words, our predictor needs to see as far into the future as the break-even threshold, that in our case, it is no more that 110 seconds, according to the maximum boot time for the machines in our cluster.

Leveraging predictive capabilities, the optimization approach can cope with wellknown patterns in measurements readings, such as trends, that indicate anticipatory conditions for triggering new optimized configurations. This is important for improving the optimization decisions, guaranteeing a better quality of service for the applications in server systems [88]. For example, turning on a new server in advance before a resource saturation occurs. As will be shown in Section 2.3.5, the applicability of our predictor in the control loop using a linear regression fit through (k = 10) past workload measurements was enough to anticipate fast load increments and improve the QoS in the system. In the following experiments, since the controller sampling period is set to 25 seconds, our controller looks ahead 4 steps to help anticipate the time to boot a new server. We plan to evaluate the robustness and accuracy of this kind of prediction and other prediction methods in future work.

2.3.5 Power and performance gains

To evaluate our optimization approach, we have carried out a set of experiments in the cluster environment described in Section 2.3. The proposed optimization formulation was implemented using the solver IBM ILOG CPLEX 11 [45], which employs very efficient algorithms based on the branch-and-cut exact method to search for optimal configuration solutions. The optimization problem worst-case execution time was less than 1 second, considering our cluster setup of 3 applications and 5 physical servers (Section 2.3). Scalability issues concerning large-scale clusters using the adopted optimization model are discussed in Section 2.4.

In the experiments, we adopted an optimization control loop of 25 seconds interval. This value was found suitable in our case, but it typically depends on the target system and workload demand characteristics, like variance. As an additional work, it would be interesting to investigate the use of two control loops with distinct intervals concerning different kinds of dynamic operations in the cluster. For example, a shorter control interval for load balancing and DVFS adaptations and a longer interval for server on/off and VM migration/replication activities.

In this work, we mainly evaluated the effectiveness of our approach by means of the energy consumption reduction and QoS violation in the cluster as compared to the Linux on-demand and performance CPU governors. The performance governor keeps all servers turned on at full speed to handle peak load and dynamic optimization is not conducted. The ondemand governor allows for managing the CPU frequency depending on system utilization, but does not include server on/off mechanisms.

The allocation scheme for the **performance** and **ondemand** governors are statically configured such that response time goals can be met under the worst-case (peak) workload arrival rate. That is, a static number of VMs were deployed and configured on each physical server for every application running in the cluster. The respective workload allocation shares for all the applications to be balanced among the physical servers in the cluster are statically defined as follows: 28% for Edison; 13% for Farad; 25% for Galvani; 26% for Gauss; and 8% for Tesla. That means we adopted a fixed weighted round-robin method for application workload balancing given the measured capacity for each server (cf. Section 2.1.1).

The experimental results for the optimization execution are given in Figure 2.9. The upper and middle plots show, respectively, the throughput and response time measured for each application in the cluster. The bottom plots show the cluster load as a function of the average CPU utilization of the currently active servers (left plot) and the cluster

Policy	Energy (Wh)	Savings	QoS violations
Performance	281.45		0%
On-demand	241.84	14.07%	0%
Optimization (reactive)	134.67	52.15%	9.72%
Optimization (predictive)	143.61	48.97%	4.79%

Table 2.3: Comparison of cluster management policies

power consumption (right plot), measured using the WattsUp Pro [25] to sample the power data every second. So, the energy consumption of the cluster could be calculated by the sum these power values over each second interval time.



Figure 2.9: Dynamic optimization execution with predictive capabilities

As shown in Table 2.3, by using our approach, the energy consumption in the cluster is substantially reduced. The main argument for these energy savings is the fact that the baseline (or idle) power consumption of current server machines is substantial. This in turn makes server on/off mechanisms (used by our optimization) very power-efficient. The *energy savings* are reduced percentages with respect to the **performance** policy. The percentage number of *QoS violations* were calculated as the sum of how many times the response time of all applications missed their deadlines divided by the total number of requests completed during the experiment. Clearly, there is a trade-off between QoS and energy minimization. Both **performance** and **ondemand** policies produced zero QoS violation. On the other hand, the energy optimization achieved by our approach lead to some QoS violations. By using a predictive optimization policy, about 7% more energy was consumed although with the benefit of an approximately 50% less QoS violations. We show that a cluster system managed using our approach is effective in reducing the power consumption costs, when compared to built-in Linux power management policies, while still maintaining good QoS levels. Our experiments are based on typical time-varying workload traces for web servers. Other power management policies adopt heuristics to turn on/off servers in a predefined order based on a power-efficiency metric for all servers in the cluster, like those described [85, 95]. To compare our approach with their work, migration/replication for the VMs and workload balancing schemes would need to be designed and implemented accordingly, which would incur in a large amount of time consuming work. We left such comparisons for future work.

2.4 Scalability concerns

To evaluate the scalability of our approach, we generated different pairs of server-application setup. For each pair, we ran the CPLEX to build and solve 180 instances. This means that each instance uses as its application demand vector the workload data at each time interval of 10 seconds using the traces shown in Figure 2.8.

The CPLEX solver was executed for every instance with a user-defined solution time limit of 180 seconds, which is related to the maximum allowed control period used in our dynamic optimization policy for managing the cluster environment. Table 2.4 shows the results of simulations with different number of servers and applications. From 5 to 30 servers, the optimal configuration solutions were found in all 180 runs within the solution time limit. From 50 to 100 servers, there is at least one instance where CPLEX could not find the optimal solution for all instances within 180 seconds.

Server,App	Avg. (s)	Stdev. (s)	Max. (s)
(5,3)	0.022	0.018	0.070
$(10,\!6)$	0.054	0.035	0.250
(15, 9)	0.062	0.038	0.240
(30, 18)	0.392	0.913	8.610
(50, 30)	13.630	29.959	180.010
(80, 48)	58.941	53.570	180.020
(100,60)	80.135	52.394	180.030

Table 2.4: Scalability simulation

In order to speed up the process of obtaining a high quality solution, we adopted a simple heuristic by setting a gap tolerance of 5% with respect to the optimal solution. This is a user-defined value and intends to allow the solver to provide acceptable solutions in a short amount of time. Table 2.5 summarizes the simulation results when the minimum gap tolerance criteria was adopted. From 5 to 350 servers, the configuration solutions were found in all 180 runs within the gap tolerance (5%) and the solution time limit (180

Server,App	Avg. (s)	Stdev. (s)	Max. (s)
(5,3)	0.006	0.008	0.040
(10,6)	0.023	0.022	0.100
(15, 9)	0.031	0.030	0.130
(30, 18)	0.062	0.067	0.540
(50, 30)	0.139	0.281	2.390
(80, 48)	0.267	0.235	3.000
(100, 60)	0.481	0.409	3.080
(200, 120)	2.893	1.993	11.550
(350, 210)	16.488	12.979	75.440
(500, 300)	48.409	41.472	181.030

seconds), with a maximum processing time about 75 seconds. For 500 servers, there were three instances for which CPLEX could not find the solution within the time limit.

Table 2.5: Scalability simulation using the optimality gap criteria

This strategy considers the solution gap between the best integer feasible solution found so far and the lower bound (LB) provided by the solver, which is usually calculated by solving a pure linear version the original problem. In minimization problems, the LB can be seen as a reference value which ensures that the optimal solution is greater or equal than this quantity. Considering the small gap value used, the CPLEX was capable of finding highly acceptable solutions, i.e., close to the optimal lower bound.

Even though we generate a number of scenarios involving different pairs of serverapplication setup, it is not possible to assume that the CPLEX will have a similar behavior in all instances. The main difficulty is that the branch-and-cut method has a worst-case exponential time complexity and depending on the combination of application workloads, this approach may lead to poor solutions in an acceptable runtime execution (solution time limit). Nevertheless, based on the simulations presented here, we have observed that the CPLEX performs well on the average case.

Given a typical optimization control period of few minutes, such as used in [63], the proposed optimization approach is suitable and scales well for clusters with up to 350 machines. This seems to be a reasonable size for a server cluster set, because, for instance, servers can be divided in smaller clusters or racks in a hierarchical fashion to address scalability issues. For example, Google uses racks with 40 and 80 PCs and racks are organized into clusters, typically with 30 or more racks [42, 43]. Typically, clusters are the basic unit of management.

As the numbers of servers and services increase in the cluster, the direct use of a MIP formulation (Section 2.2.1) to obtain integer feasible solutions for the cluster configuration problem becomes prohibitive, due to its large number of variables and constraints in the model. To make our approach practical for (very) large-scale heterogeneous server clusters,

we can rely on the following optimization heuristics efforts aimed at providing high quality solutions in short amount of processing time.

In [60] we described a reformulation of the cluster configuration problem using a Dantzig-Wolfe Decomposition [22]. We developed a column generation based heuristic, called Rounding Heuristic (RH), for obtaining a feasible integer solution by rounding up the value of the fractional variables of a relaxed optimization solution. This proposed solution approach was found highly effective in terms of very low optimality gap and computational time.

We also proposed a Multi-Start Iterated Local Search for large-scale packing problems with multiple resources. This method iteratively applies a local-search procedure to improve the solutions, along with shaking moves to escape from local optima. To reduce the computational time, shaking moves/restarts are dynamically triggered whenever the improvement is estimated to be too small with respect to the total solution cost. Extensive experimental evaluations demonstrate the remarkable performance of the proposed method on cluster configuration with up to 50,000 processes and 5,000 machines. More details can be found in [73] and [96]. Note that the integration of these research works in our optimization framework is orthogonal to the current work and left for future work.

2.5 Related work

Optimization approaches, based on the bin packing problem, for configuring virtualized servers are described in the literature, such as [12, 54]. However, their models are not designed for power-aware optimization. In [98], the authors present a two-layer control architecture aimed at providing power-efficient real-time guarantees for virtualized computing environments. The work relies on a sound control theory based framework, but does not addresses dynamic virtual machine allocation or migration and machine on/off mechanisms in a server cluster context. A heuristic-based solution outline for the power-aware consolidation problem of virtualized clusters is presented in [91], but it does not provide a means to find solutions that are at least near to the optimal.

The non-virtualized approach described in [26] determines predefined thresholds to switch servers on/off (given by simulation) based on CPU frequency values for the active servers that must match in order to change the cluster configuration to meet the performance requirements. However, their proposal does not provide an optimal solution as a combination of which servers should be active and their respective CPU frequencies. The problem of optimally allocating a power budget among servers in a cluster in order to minimize mean response time is described in [33]. In contrast to our approach, which is designed to minimize the cluster power consumption while meeting performance requirements, their problem poses different optimization goals and does not address a virtualized environment considering multiple services in a server cluster.

A dynamic resource provisioning framework is developed in [63] based on lookahead control theory. Their approach does not consider DVFS, but the proposed optimization controller addresses interesting issues, such as switching machines costs (i.e., overhead of turning servers on and off) and predictive configuration model. Our approach also copes with switching costs and it includes the ability to incorporate prediction techniques in the optimization strategy to improve the configuration decisions. A power-aware migration framework for virtualized HPC (High-performance computing) applications, which accounts for migration costs during virtual machine reconfigurations, is presented in [95]. Similarly to our approach, it relies on virtualization techniques used for dynamic consolidation, although the application domains are different and the algorithm solution does not provide optimal cluster configurations.

Contrasting with [63, 91, 95], our approach takes advantage of dynamic voltage/frequency scaling (DVFS) mechanisms to optimize the server's operating frequencies to reduce the overall energy consumption. An approach based on DVFS is presented in [44] for power optimization and end-to-end delay control in multi-tier web servers. Related approaches, such as presented in [9, 17, 41, 51, 55, 85, 89] also rely on DVFS techniques and/or include server on/off mechanisms for power optimization. However, these approaches are not designed (and not applicable) for virtualized server clusters. That is, they do not consider multiple application workloads in a shared cluster infrastructure.

Predictive/proactive optimization policies have been shown to avoid unnecessary and disruptive configuration changes due to workload fluctuations, and thus may provide further energy reduction and better quality-of-service provided by the applications in a server cluster [63, 88]. Although our approach is not meant to address in detail the specific aspects on workload prediction, it allows for including predictive capabilities in our optimization control loop during monitoring activities.

The idea of energy proportionality, presented by Luiz Barroso and Urs Hölzle in [5], is that computing systems should consume power in proportion to their utilization level. The energy-proportionality concept would enable large energy savings in servers, considering their study at Google that servers in data centers are loaded between 10 and 50 percent of peak, with a CPU utilization that rarely surpasses 40 percent. Although power proportionality is very important, it does not reduce the importance of ensuring that data center resources should be near fully utilized, as pointed out in [39]. Thus, our solution to this problem is to efficiently manage the cluster utilization leveraging server virtualization and CPU DVFS techniques.

Some proposals like FAWN [1] explore the idea of building clusters of low-power embedded devices coupled with flash storage, which operate efficiently for specific I/O bound workloads. There is also an interest in distributing the workload across server clusters in different locations with respect to their energy consumption [64, 82]. An issue here is that energy cost models for distinct time zones and variable electricity prices need to be specified accordingly.

2.6 Summary

We have applied our approach to power optimization in virtualized server clusters, including power and performance models, an optimization MIP model and a strategy for dynamic configuration. In the optimization model, we addressed application workload balancing and the often ignored switching costs due to frequent and undesirable turning servers on/off. The major goal of this work was to develop and demonstrate the feasibility of our optimization approach for power and performance management, while providing experiments with realistic Web traffic driven by time-varying demands with real system measurements.

Our experiments show that our strategy can achieve energy savings of 14% compared to DVFS-enabled cluster (Linux on-demand kernel governor) and 52% compared to a typical uncontrolled system (all CPUs running at maximum speed), while simultaneously meeting applications' performance goals as specified in their response time deadlines. Moreover, many power management schemes like [9] and [41] would need extensive power and performance measurements, which should be repeated every time upon new installations, server failures, upgrades or changes. The power and performance models presented in this work can simplify such benchmarking activities.

Chapter 3

Thread assignment optimization for heterogeneous multi-cores

One important challenge that arises in this scenario is to determine the most suitable thread assignment decision, while meeting soft real-time requirements for the application threads and reducing energy consumption in the heterogeneous multi-core system. Considering that different threads can have different runtime requirements, improvements in energy efficiency and performance can be achieved by assigning each thread to run on the core type (large or small) that is best suited for it. Such thread assignment decisions should be re-evaluated at runtime since a thread may have execution phases with different performance demands over time. The optimal energy-efficient thread assignment has to satisfy the threads' computational demands. Assigning a thread to a large core rather than to small core might be counter intuitive for energy-efficiency, because a large core consumes more power than a small one. However, executing a particular thread for a while on a large core can shorten the overall execution time, and hence might end up consuming less energy.

Previous studies have shown that the performance of a thread can vary considerably depending on the memory-intensiveness of co-scheduled threads in the multi-core system [15, 69]. Typically the running threads have working sets larger than the on-chip caches of their allocated core and may impose a significant burden on the limited off-chip memory bandwidth, shared among all cores. Some threads that make heavy usage of the shared cache will eventually evict the data of others threads, possibly leading to increased cache misses and contention during the execution of those threads. The most suitable thread-to-core assignment decision comes with a challenging trade-off: moving the least memory-intensive thread from a big to a small core may help alleviate shared resource contention, lowering thread instruction throughput/retirement rate, but this could considerably slowdown its performance and increase energy consumption.

For the above described context, we tailored our optimization approach to determine



Figure 3.1: Heterogeneous platform

an optimized mapping of threads to a set of available cores to reduce energy consumption while meeting soft real-time performance requirements for a set of application threads. We propose an ILP (Integer Linear Programming) optimization model associated with a periodic thread re-assignment scheme. The optimization model is solved at runtime given updated values of key performance counters, such as IPC and LLC miss rate, aiming at characterizing the different execution phases of running threads on different core types.

In a multi-core heterogeneous platform, as the one depicted in Figure 3.1, application threads can take advantage of large cores for their CPU-intensive execution phases, while memory-intensive threads are best suited and assigned to small cores [29]. Therefore, it is important to effectively characterize the resource demands of threads such as CPU and memory bandwidth needs/requirements. We take advantage of specific counters provided by the hardware to determine the performance characteristics for the running thread, namely the IPC (Instructions committed Per Cycle) or IPS (Instructions Per Second)¹ as a measure of the CPU load and LLC (Last Level Cache) misses as a measure of the memory load in a given interval. The LLC miss rate is known to be a good measure to characterize memory-intensive threads, because there is a strong correlation of threads having a high LLC miss rate and the high demand of memory requests [15].

Previous work has used mainly one of these counters, assuming that there is a correlation between them [59, 86]. Figure 3.2 shows the IPC and LLC misses for the *astar* SPEC CPU2006 benchmark on an Intel Core2 processor. We can clearly observe the distinct phases of high/low LLC misses and low/high IPC. Considering the overall low IPC measure, *astar* can be classified as a memory-intensive program [47]. However, we can notice that it exhibits a CPU-intensive phase for 200 billions cycles in the middle of its execution that should be properly taken into account. We can see that the correlation is

¹The IPS measure is more adequate when cores differ in clock frequency and cache size, possibly having the same micro-architectural characteristics.

not as strong as one-to-one, contrasting the IPC and LLC misses between 0 to 100 billions of cycles and 300 to 400 billions of cycles. In order to consider this type of behavior, in our approach, we use additional performance metrics independently as inputs for our optimization model. In particular, we use two performance measures aimed at augmenting thread assignment optimizations. Our optimization scheme achieves performance gains and energy savings over purely static assignment and related scheduling schemes that do not take into account combined computational and memory bandwidth requirements.



Figure 3.2: Time-varying demands of IPC and LLC misses for astar benchmark running on Intel Core2

We have implemented a prototype of our thread assignment scheme at user-level leveraging Linux scheduling and performance monitoring capabilities. We evaluate our approach on a real 64-bit quad-core x86 processor, where frequency scaling was used to emulate heterogeneous cores. The assignment of threads to cores, as specified by our optimization scheme, is implemented via a system call in the Linux scheduler (CPU affinity) that places a thread on a specified core at runtime. To implement our approach in the real system, we propose a regression model to predict at runtime the performance of threads running on different core types.

3.1 Optimized thread assignment

In this section we present our approach for solving the energy-aware thread assignment problem accounting for real-time performance and memory constraints of the running threads in heterogeneous multi-core systems. Our scheme is novel in three main aspects, in comparison to existing solutions. First, our scheme relies on performance counters to monitor the execution behavior of a set of threads and determine an optimized thread-tocore global assignment, instead of performing local thread assignment decisions. Second, the optimization objective function deals with combined energy savings and performance gains by meeting soft real-time computational and memory bandwidth requirements in the heterogeneous multi-core system. Third, our scheme explicitly uses two performance measures, instruction retirement and LLC miss rate, rather than relying on their possible/probable correlation.

3.1.1 Optimization problem

We deal with the multi-core optimization problem of determining the most efficient thread-to-core assignment that minimizes energy consumption while meeting temporal requirements. In our formulation, we divide the temporal requirements into computational and memory requirements. Our optimization problem is defined as follows. Let $N = \{1, 2, ..., i, ..., n\}$ be the set of core types in the system with maximum g_i cores for each type *i*. It is predicted that *n* will be 2 for the near-future multi-core systems [59, 92], since two distinct core types combined with core on-off capabilities are able to capture much of the benefits from heterogeneity. The multi-core system has a memory controller with bandwidth capacity *B*, which is measured in requests per second, where each request has a fixed size (number of bytes).

In our model, each core of type $i \in N$ has exactly the same characteristics, having computational capacity C_i , given by MIPS (Million Instructions Per Second), and power consumption given by P_i , which includes both static and dynamic power. We use MIPS instead of IPC, since MIPS is a more adequate metric when cores can differ in clock frequencies, possibly having the same micro-architectural characteristics. In fact, we use a multi-core system with cores running at different clock speeds to emulate heterogeneity in our implementation (see Section 3.4). We assume that unused cores can be quickly powered down, rather than left idle, meaning they have no idle power consumption. For example, modern Intel processors rely on effective power gating techniques to quickly shutdown power to the idle cores [46].

The set of application threads is denoted by $K = \{1, 2, ..., m\}$. The performance demand of each thread $k \in K$ running on a core type $i \in N$ requires a computational rate (in MIPS)² given by c_{ik} and a memory access rate (in requests per second) denoted by b_{ik} . Our real-time constraints are soft in that they can be violated without severe performance loss to the system; in that sense, attempting to satisfy the rate of computing and memory requests is sufficient to satisfy the deadline constraints as well (see more in Section 3.3.2 where we measure how well the system did in completing the requests on a timely fashion [23]).

The optimization problem is formulated as an Integer Linear Program (ILP), described

 $^{^{2}}$ Even though the rate is expressed in millions of instructions *per second*, the actual period of the applications varies and is not necessarily one second.

in Equations (3.1)-(3.5). The decision variable is given by x_{ik} , which is a binary variable to denote whether or not a thread $k \in K$ is assigned to core type $i \in N$.

$$Maximize \sum_{i \in N} \sum_{k \in K} \left(\frac{c_{ik}^{\gamma}}{P_i}\right) x_{ik}$$
(3.1)

Subject to

$$\sum_{k \in K} c_{ik} x_{ik} \le C_i g_i \qquad \forall i \in N \tag{3.2}$$

$$\sum_{i \in N} \sum_{k \in K} b_{ik} x_{ik} \le B \tag{3.3}$$

$$\sum_{i \in N} x_{ik} = 1 \qquad \forall k \in K \tag{3.4}$$

$$x_{ik} \in \{0, 1\} \qquad \forall i \in N, \ \forall k \in K \tag{3.5}$$

The ILP model describes the problem of assigning threads to core types respecting computational and memory bandwidth requirements. The objective function given by (3.1) aims to find a mapping of threads to core types (x_{ik} values) that optimizes thread performance and energy consumption in the multi-core system. Constraints (3.2) guarantee that the total thread computational workloads do not exceed the total processing capacity of available cores for a given core type, obeying the soft real-time requirements of the application. Constraints (3.3) ensure the total memory workload of threads (allocated to certain core types) does not exceed the available memory capacity, avoiding stalls and contributing to the soft real-time requirements of the application. Constraints (3.4) require that each thread is assigned to a core type. Constraints (3.5) define the domain of the problem variables.

In the objective function, the metric used for combining thread performance and energy efficiency is weighted performance per watt, namely $MIPS^{\gamma}/Watt$. The constant γ is a power-aware design parameter that gives a smoothing or boosting performance effect in the system. This usually requires a trade-off between the performance and energy consumption. To optimize a metric of performance per watt we can set $\gamma = 1.0$. A value of $\gamma = 2.0$ is more suitable to allow further emphasis on performance gains: the objective function minimizes (in fact, maximizes the inverse) of the energy delay product per instruction, given by $Watt/IPS^2$, which can be rewritten as $(Watt \times S \times S)/I^2 =$ $(Energy \times Delay)/I^2$. This objective function aims to minimize both the energy and the amount of time required to execute thread instructions [16].

Figure 3.3 shows a comparison of the energy efficiency metric given by $MIPS^2/Watt$ when running the same *astar* SPEC benchmark on different cores types. As can be seen, the efficiency metric varies over different execution phases between the core types and will



Figure 3.3: Performance and energy efficiency over time during a given thread execution on different core types

be used by our approach for guiding dynamic thread assignment decisions. Note that the graph in Figure 3.3 shows that in the middle of the execution (approximately at 43% of the thread execution lifetime) there is no clear winner with respect to energy efficiency and the thread assignment decision has to be made in some other way. In fact, using our optimization model, the decision will be influenced by the memory demands and energy efficiency of the other threads in the system. A thread may have almost the same energy efficiency but different LLC miss rate when running on either large or small cores. Our optimization model will move threads between core types to optimize energy efficiency and avoid exceeding memory constraints in the system.

3.1.2 Thread performance prediction

One important practical issue in heterogeneous multi-core systems is the ability to predict the performance behavior (IPS and LLC misses) of a thread currently running on a given core type when assigned to a different core type. Previous works [61, 8] require each thread to run on both core types to be able to determine the IPS ratio between large and small cores. As noticed in [90], such a direct IPS measurement on both core types poses many practical issues and incurs much overhead. In contrast to those works, we addressed the thread performance prediction by collecting performance data from a representative set of workloads in the system and establishing the following linear regression model:

$$\begin{split} IPS_{small} &= w1*IPS_{large} + w2*LLCM_{large} + w3\\ IPS_{large} &= w4*IPS_{small} + w5*LLCM_{small} + w6\\ LLCM_{small} &= w7*IPS_{large} + w8*LLCM_{large} + w9\\ LLCM_{large} &= w10*IPS_{small} + w11*LLCM_{small} + w12 \end{split}$$



Figure 3.4: Predicting thread MIPS and LLC miss rate from large to small core

The above performance prediction model for our multi-core system is derived from offline performance data running the threads of the SPEC CPU2006 benchmark suite individually on each core type. The coefficients w1 to w12 are derived from running the machine learning software Weka [38]. A 10-fold cross-validation of the performance model yielded a coefficient correlation of approximately 97% for the least squares fitting to the collected performance data.

In our Linux implementation in Section 3.4, we use the equations of the prediction model at runtime to obtain the values for the computational c_{ik} and memory b_{ik} demands, for each core type $i \in N$ running thread $k \in K$. Our prediction model does not estimate the values for the next reassignment interval on the same core type; we use real measurements for those values. These measured or computed values are used as inputs to the ILP model. We use the offline performance data directly in our simulation methodology, as described in Section 3.2, instead of relying on the prediction model.

Figure 3.4 illustrates the prediction of MIPS and LLC miss rate for the tonto SPEC benchmark when running on a large core and intended to run a small core. That is, the data were collected from a large core to predict the performance on a small core. The graphs in Figure 3.4 show predicted versus actual values on a small core. We observe that our prediction model could successfully distinguish low, mid and high compute-intensive thread phases in most cases. More importantly, for practical purposes minor inaccuracies in our prediction model did not impede our approach from achieving performance and energy improvements.

Note that thread/core performance characterization needs to be done only once at the design stage. Given a set of representative workloads intended to run on the multicore system, there is no need to recalibrate the parameters of the prediction model. Nevertheless, when substantial changes are observed in the behavior of the workloads, better results can be obtained by updating the coefficients of the prediction model or determining the coefficients at runtime.

3.1.3 Solution to the optimization problem

We solve the ILP model to determine the best assignment decision of threads to core types. Our ILP model is implemented using the Gurobi solver [36]. We set heuristic emphasis and solution time limit in the solver to provide (near-)optimal solution in a reasonable amount of time and minimize the overhead of our approach. Clearly, increasing the solution time bound would improve the assignment solution, however we found that a time limit of few milliseconds is able to provide on average 1% percentage of distance from the optimal solution (see Section 3.3). With time limits imposed on the solver for optimization solution, the solver reports the best feasible solution found at the end of the time limit. Currently, when no feasible solution can be found within the time limit, our scheme just keeps the configuration found in the last optimization call.

To allocate each thread to an available core of a given assigned core type, several packing heuristics can be used, such as first-fit, best-fit or next-fit decreasing. We use a next-fit (round-robin) strategy that first sorts the threads in decreasing order by core performance demands and then places a thread in the next available core that has sufficient remaining capacity to accommodate the thread demand (see Section 3.4). In case more than one thread is assigned to a given core, we assume that the underlying OS allows to time-share the core among different threads. Actually, the thread allocation phase is applied only when threads have to move between different core types, preserving as much as possible the existing assignment of threads to cores to take advantage of caches current contents.

During the thread allocation phase, a given thread workload may have to be scheduled to different cores in order to fully utilize the multi-core system. Figure 3.5(a) shows a scenario where threads A, B, and C are best mapped to a given core type. Given this assignment solution, we rely on a multi-processor OS scheduler to enable the effective thread execution among cores. Basically, this can be accomplished by migrating part of a thread workload during scheduling activity [10], such as depicted in Figure 3.5(b).

In practical terms, the ILP assumes that context switch overhead for different thread executions is relatively small [92], when multiple threads are supposed to run on a given core. In this work, we deal with the problem of assigning threads to core types respecting computational and memory bandwidth requirements. We rely on underlying scheduling techniques developed elsewhere [10] for the exact scheduling implementation, or even simultaneous multi-threading (SMT) support found in modern multi-core systems.



Figure 3.5: Optimization solution: (a) thread allocation phase and (b) example of scheduling execution

3.2 Simulation methodology

We developed an in-house simulator, similar to [92], aimed at comparing our approach against other related schemes using collected performance data. In the simulation, we consider a heterogeneous multi-core system where two types of cores are available and can differ in power, performance, and architectural characteristics, such as in-order vs out-of-order execution, maximum clock speed and cache size [92]; also, the area size of a large core is equivalent to three small cores (3:1 ratio) [59]. Thus, the basic multi-core system is composed of one large and three small cores, where $N = \{1, 2\}$ and $g_1 = 1$ and $g_2 = 3$ in our ILP model (Section 3.1.1).

In our simulated multi-core system, large cores have characteristics based on Intel Core2 processor, whereas small cores are based on Intel Atom processor. Those core types were chosen because they differ significantly in terms of power and performance characteristics but have the same Intel 64-bit ISA (instruction set architecture). More precisely, Intel Core2 operates at 2.8Ghz and has a 4-way super-scalar and out-of-order architecture with 3M L2 cache size, while Intel Atom's architecture has a 2-way issue and in-order execution with 1.66 GHz clock speed and 512k L2 cache size. We assume 30% of total chip power is consumed by the "uncore" part, that is, last level cache (LLC), memory controller, and I/O controller; the approximately per-core power consumption of a small core (Intel Atom) is 3W and large core (Intel Core2) is 23W [46].

3.2.1 Thread execution behavior

We use the *perfmon2* Linux interface [27] to gather thread execution traces, consisting of IPC (instruction per cycle) and LLC misses, where each thread represents an instance of a SPEC CPU2006 benchmark. The *ref* input option is used in all the benchmark

programs. The *perfmon2* interface allows for monitoring hardware performance events on a per-thread basis. We set a sampling time of 50ms to collect data of each thread [92]. From running the benchmarks alone on each core type, we obtained the values for c_{ik} and b_{ik} that were used as inputs to the ILP model, in Constraints (3.2) and (3.3), respectively.

In heterogeneous multi-core systems, a thread has different performance on each core type, mostly due to the distinct micro-architectural characteristics found in those core types, such as clock speed, private cache size, in-order vs out-of-order execution, and branch predictor capabilities. Based on analysis of the collected execution traces for different programs from the SPEC CPU2006 benchmark suite, we observed a correlation between IPC or LLC misses and thread execution phases [8, 61, 59], which allows to explore a dynamic thread assignment scheme based on IPC or LLC miss rate indicators. In our work, we go beyond that, characterizing the thread behavior in terms of both CPU and memory requirements, because we have found that there is not always a clear one-to-one correlation between these two metrics in the entire execution of the threads. We explicitly use these two performance measures to improve thread assignment optimizations.

3.2.2 Core and memory system performance

We determine the capacity of the hardware with respect to the number of instructions and number of memory requests that can be processed. Based on the execution of a high compute-intensive thread, we derive the processing capacity for each core type as follows. We measured the highest IPC for each core type, by running all the SPEC CPU benchmarks one thread at a time. Multiplying the highest IPC obtained by the clock speed, we can estimate how many instructions can be executed in one second.

The results are as follows: small core type maximum IPC is 1.44 and large core is 2.71. Thus, the core capacity is given by $C_1 = 1.44 \times 1.66Ghz = 2,390$ MIPS for small cores and $C_2 = 2.71 \times 2.8Ghz = 7,588$ MIPS for large cores. The memory bandwidth B for the multi-core system is set to be 52 million requests per second, which is based on real measurements of a heterogeneous platform configuration composed of large and small cores [92]. The calculation assumes a memory bandwidth capacity of approximately 3.3GB/s with fixed requests of 64 bytes. Those measurements are used as inputs to the ILP model, in Constraints (3.2) and (3.3) in Section 3.1.1.

3.2.3 Simulator environment

Our simulation is driven by a workload trace, where each entry corresponds to the number of committed instructions I(i, k), number of cycles (including stall cycles) C(i, k) and LLC misses L(i, k) collected over a fixed monitoring period (P = 50ms), for each combination of thread k and core type i. The workload trace is constructed based on thread performance measurements collected in a real system (see Section 3.2.1). The performance traces were collected running a single thread at a time. In our simulation, we ignore the effect of context switching even when more than one thread runs on a given core because it has been shown that this is relatively small [61, 92].

The simulation interval of 50ms is divided in two main phases: (1) thread monitoring/execution and (2) thread reassignment. The monitoring/execution phase is performed by tracking the workload trace and determining the execution time for each thread running on a given core type. The thread reassignment phase is executed just after the completion of the monitoring/execution phase. At each reassignment period, the values of IPC and LLC misses for each thread are computed based on the last simulation interval. Using our approach, the reassignment phase consists of solving the optimization model and deciding an energy-efficient thread-to-core mapping that will take effect in the next simulation interval. Each entry of the workload trace is used as input data to the optimization model to specify thread computational and memory requirements. Also, with this simulation infrastructure, we can implement different dynamic thread assignment schemes to compare with our optimization approach (see Section 3.3).

We use the following values as inputs for the simulations: (a) migration overhead is 450 microseconds, as measured by [92]. Since P = 50ms, the migration overhead is at most 0.9% = 0.450 ms / 50 ms; and (b) a single memory request latency (penalty due to LLC miss) is on average 75ns, based on real measurements [92]. The energy consumption of a thread, accumulated over each simulation interval, is obtained by multiplying the simulation interval length by the dynamic power consumption of the core type allocated to the thread. The dynamic power measure is given by the thread utilization multiplied by the core peak power, where the utilization of a thread means its computational demand in a given interval divided by the allocated core capacity.

3.2.4 Estimating thread execution time

To evaluate whether the thread k is meeting its soft real-time requirements, we need to determine the execution time of the thread. For that, we use the number of core execution cycles C(i, k) and LLC misses L(i, k) that the thread will perform on its assigned core type i at core clock rate R(i). We assume that each memory request is due to a LLC misses and that it takes M(i) cycles on a core type i. However, the LLC misses may stall the thread a variable amount of time, depending on the contention at the memory controller. Below we show how we calculate thread execution time over each simulation interval.

We account for potential contention on the limited bandwidth of the memory controller by including the following factor in the thread execution time. We sum up the LLC miss rates L(i, k) of all threads k in the current simulation interval. In case the sum, S, does not exceed the bandwidth of the memory controller, B, that is $S/B \leq 1$, the thread execution is not affected by memory contention; otherwise, when S/B > 1, the memory contention for each thread k is given by $MC(k) = (max\{1, S/B\} - 1) \cdot L(i, k) \cdot M(i)$. The execution time of a thread k, running on core type i, is modified as follows at each simulation interval: E(k) = [C(i,k) + MC(k)]/R(i). Note that this is very conservative, since it assumes every LLC miss will add the full penalty of a memory access to the execution time, while in practice multiple-issue memories and pipelining mitigate this penalty. Being conservative is commonly done in real-time systems, where timeliness is an important issue. Note that the number of LLC misses is typically much lower than the capacity of a core, since the cores would stall if the threads were issuing too many memory requests.

In addition to memory contention, our simulation framework also considers thread migration between cores. In that case, we add a constant ML (migration latency) to the execution time: E'(k) = E(k) + ML. Since both core types used in our work (Atom and Core2) have the same Intel 64-bit ISA, we use the accumulated number of committed instructions A(k) to keep track of the execution of each thread k. When a thread resumes on a different core type after a thread re-assignment, the simulator resumes the thread execution from the point given by A(k), since the ISA is the same for both core types.

After determining the core where thread will run in the next interval, the execution time calculation allows us to determine whether the execution phase of thread k will expect a slowdown. This happens in case of memory contention (when MC(k) > 0) or migration latency (when ML > 0); in those cases, we add a factor given by: SD(k) = $min\{0.9, (MC(k)+ML)/P\}$, where P is the simulation interval. We set a slowdown limit of 90% because we assume a thread makes at least 10% rate of progress in its execution, given that the MC may be overestimated in our model. The slowdown factor SD is used to update the instruction pointer A(k) for the next simulation interval for each thread on both core types. We assume that within a given simulation interval, the CPU instructions I(i, k) and memory request L(i, k) are uniformly distributed, so that we can easily map the value of each measure proportionately to a time interval in a given simulation interval.

3.3 Simulation results

We have carried out a set of simulation experiments, where the workloads are given by a set of combinations of individual programs collected from running the SPEC 2006 benchmark suite (see Section 3.2.1). We adopt a similar workload configuration to the one described in [59], where the number of threads is equal to the sum of cores in the system and each workload is composed of a set of programs of the SPEC benchmark as follows. The workloads A1 to B4, as shown in Table 3.1, are composed of three instances of one SPEC benchmark and one instance of a second benchmark. These workloads are divided into two groups: workloads A1 - A4 have high variability on IPC ratio (large-to-

Workload	3-instances — IPC ratio	1-instance — IPC ratio
A1	soplex - 1.74	gamess -4.14
A2	milc - 1.75	zeusmp - 3.53
A3	astar - 1.90	calculix - 3.82
A4	lbm - 1.52	bwaves — 3.06
B1	omnetpp - 2.27	gobmk - 2.31
B2	bzip2 - 2.41	cactusADM - 2.49
B3	mcf - 2.76	namd - 2.92
B4	GemsFDTD - 2.76	sphinx3 - 2.94

small core) and workloads B1-B4 have medium/low IPC variability between the selected benchmark programs.

Table 3.1: Workload configuration

We compare our scheme with three other schemes: (a) **static** assignment; (b) **IPC**driven algorithm [8]; and (c) **bias** scheduling [59]; see Section 3.5 for more details on the latter two algorithm. The static assignment is obtained by running the ILP once at the beginning of the simulation, based on the average IPC and LLC misses of the entire execution of the benchmarks. For the IPC-driven and bias scheduling algorithms, the simulator determines the relative threads' IPC ratio or bias (characterized by LLC misses). A large core bias would represent lower LLC misses. Threads with the highest IPC ratios (or lowest bias) are mapped to the available large cores, whereas the lowest IPC ratios (or highest bias) are mapped to small cores. In our scheme, the IPC and LLC misses collected over the simulation are used to specify thread computational (in MIPS) and memory (in requests per second) requirements and yield an optimized thread assignment solution. In this work we are interested in the behavior analysis of the thread assignment decisions. We ignore the known measurement overheads of the IPC scheme [29] and we assume a priori knowledge about the exact workload values of each thread running on each core type.

In our ILP model we use a performance constant $\gamma = 2.0$ which is related to the energydelay product [16] (see Section 3.1.1). During thread execution phases with high memory contention, we observed that our model may lead to infeasible solutions because of memory bandwidth constraint violation. To address this issue, when the solver identifies that the memory constraint is violated, we solve a new instance of the ILP model by performing two modifications at the same time: (1) relaxing/removing the memory constraint and (2) decreasing the value of the performance parameter γ . With those modifications in the ILP model, the solver intends to find an assignment solution that decreases the bandwidth demands by placing more emphasis to using small cores (recall that smaller cores issue fewer memory requests per time unit). However, a good optimized solution should balance the usage of large and small cores to avoid negative impact on the overall performance. For the set of workloads used in this work, we empirically tested values for γ from 1.0 to 2.0 spaced by 0.1 and observed that keeping the value close to 2.0 caused higher contention overhead and setting lower values close to 1.0 led to higher performance loss. The best parameter found, in terms of energy-delay, was $\gamma = 1.4$. Each subsection below analyzes the simulation results for different metrics, namely energy-delay product, tardiness, memory bandwidth contention, and optimization execution overhead.

3.3.1 Energy-delay product

We first evaluate our approach in comparison with related schemes by computing energydelay product. Using a similar methodology as described in [59], we report the execution time and energy consumption of the first run for each thread. We multiply the execution time by the energy consumption to derive the *energy-delay* product (EDP), which quantifies the effectiveness of trading energy for performance. We avoid idle cores in the system by allowing a thread to restart its execution as soon as it is finished. We report the workload EDP by summing the individual EDP of all threads.

The energy-delay product normalized to the best static scheme for the workloads A1 to B4 is shown in Figure 3.6. Improvements of dynamic schemes over static assignments are expected because a static scheme cannot cope with thread phase changes, as already observed in previous works [61, 8]. Surprisingly, the bias scheduling showed, on average, practically no improvement over the best static scheme. We observed that in general the bias metric, used for characterizing the thread behavior, led to very few thread migration activities when compared to the IPC algorithm and our scheme. Our results demonstrate that avoiding thread migration is not essential, because migration can provide more performance benefits than its associated execution overhead and slowdown.

The IPC-driven algorithm showed good results and, for instance, outperformed our approach by 4% in the workload B2. However, considering all workloads executions, our scheme showed an average improvement over the static, IPC, bias schemes of 23%, 22%, 22%, respectively, for A workloads, and 43%, 17%, 45% for B workloads. A lower improvement over the bias and static assignment with A workloads was expected since these workloads have a high IPC ratio (from large-to-small core type) and CPU vs memory intensiveness distinction between the selected benchmarks, favoring static or less dynamic assignment schemes.

3.3.2 Tardiness

We used a real-time metric called *tardiness* [23] for each workload execution and assignment scheme. The deadline for each thread is half its running time in a small core type. The average tardiness of a thread execution is accumulated over each completed thread instance and is given by its completion time divided by its deadline, normalized by the



Figure 3.6: Total energy-delay products normalized with respect to the static scheme for workloads with varied IPC (A1–A4) and with uniform IPC (B1–B4).

number of deadline misses counted for each thread instance. Tardiness is zero when there is no deadline misses. As can be seen in Figure 3.7, our scheme always achieved smallest tardiness among all tested schemes over all workloads executions.

As expected, the static scheme had the most deadline misses and tardiness, given the tight deadlines for the threads running on three small cores. It was surprising, though, to see that both IPC and Bias had such high tardiness for the high variability workloads (A1 - A4). The average tardiness reduction was 74%, 61%, 67% over static, IPC and bias, respectively. Those workloads are composed of one compute-intensive and three memory-intensive benchmarks. The IPC algorithm aims to maximize performance without considering the actual memory intensity of the threads. This can lead to memory contention and tardiness when moving a thread to the large core that violates memory bandwidth. The Bias algorithm is aware of memory intensity and moves to a large core the thread that experiences least memory stalls. However, on average, this leads to performance loss since others thread cannot benefit from running on the large core and making progress.



Figure 3.7: Average tardiness during execution of each scheme using workloads A1–A4 and B1–B4.

3.3.3 Memory contention analysis

To observe the effects of memory contention, we executed the same workloads A1 to B4 in a system with reduced peak memory bandwidth. In particular, we reduced the memory bandwidth of the system by half, that is, the memory controller is able to handle 26 millions of requests per second. We show in Figure 3.8 a comparison of our approach with the IPC scheme. We only show a comparison with IPC because it outperformed both static and bias schemes in all executions. In the figure, we show the energy-delay product as well as normalized memory contention, that is, the number of simulation intervals when total number of memory requests of all threads exceeded the capacity of the controller, divided by the total number of simulation execution intervals.

We observe two main results from Figure 3.8. First, considering all those workloads, the average reduction on memory contention by our scheme is approximately 13% in comparison to the IPC scheme, showing that measuring and taking both the memory and CPU requirements into account is beneficial. Second, we observed a correlation between memory contention and energy-delay values: the higher memory contention is, the higher energy-delay is for different workload executions. An exception is workload A1, where our optimization scheme could minimize the energy-delay despite creating more memory contention. This can be explained by the fact that ultimately our scheme could better cope with thread phase changes and outperformed in terms of overall energy/performance benefits. By performing about 51% more migration activities in comparison with the IPC scheme, our scheme could reduce the total execution time by 40% although consuming



Figure 3.8: Memory contention analysis: comparison between our optimization and IPC scheme using workloads A1 to B4. The energy-delay products are normalized by the IPC scheme.

15% more energy and incurring in 48% more memory contention. This shows that an effective thread assignment scheme has to account for the various trade-offs, addressing all those previous aspects together, to achieve good overall energy/performance gains.

3.3.4 Scalability of optimization scheme

For our optimization scheme to be practical, the proposed ILP model must execute for a short time compared to the period used for dynamic thread assignment (in our case, 50ms). We have used the Gurobi optimizer version 4.5 [36] to solve the ILP model using one large core (Intel Core2). To evaluate the scalability of our approach, we perform some experiments regarding the resolution time for different instances of the optimization problem. Note that the overall complexity of the optimization model is dominated by the number of threads in the system, since the number of core types is typically small and fixed.

To improve the ILP solving strategy, we have adjusted some parameters in the Gurobi solver. The first change was to place more emphasis on heuristics to allow the solver to find feasible solutions quickly. For small instances (four threads), we were capable of running the Gurobi solver with a solution time limit of 1 ms; but on average these instances actually required 212 μ s with standard deviation of 75 μ s. An interesting aspect of the Gurobi solver is that it automatically uses the previous solution as an efficient warm start for optimizing a new modified optimization model. This helps reduce the solution time of subsequent optimization calls.

For medium/large instances we used a callback function to stop the solver as soon as an integer feasible solution was found. In the following results, we compare this heuristic strategy with running the Gurobi solver without time limit and without callback termination in terms of the time required to solve the model and quality of the solutions found. As above, we compare with the IPC-driven algorithm but not with static and bias schemes, because IPC outperformed both in all previous simulations.

We generated new, more demanding workloads (Table 3.2) with higher number of threads from workloads A1 to B4 (Table 3.1), but not using thread instances more than once. To perform simulations with sizeable multi-core systems, we scale the previous multi-core system by a multiplication factor to build k-large, 3k-small systems, for k = 4, 8, 16, 32. We assume that the number of threads is equal to the sum of cores in the system.

Workload	Thread composition	Threads	Core setup
С	Each instance from A1 to B4	16	4L12S
D	C + C	32	8L24S
E	D + D	64	16L48S
F	E + E	128	32L96S

Table 3.2: Workload configuration for large multi-core setups

Table 3.3 shows the results of simulations varying the number of threads and cores in the system. The **EDP Improvement** column gives the percentage improvement over the IPC-driven algorithm on each workload. The **Runtime** column corresponds to the average / standard deviation of the processing time required to solve all instances of the optimization model associated with each workload and multi-core configuration. The **Solution Gap** column gives the relative difference between the lower and upper solution bounds, that is, the best known integer feasible solution and the best bound on the solution quality of the original problem, whose initial value is usually provided by a linear programming relaxation (assuming all variables to be continuous in the model). This is a common measure for the quality of a given optimization solution that means how far at most a solution can be from the optimal one.

Workload	EDP Improv. (%)	Avg. / Stdev. Runtime (ms)	Sol. Gap (%)
С	18%	0.560 / 0.03	0.02
D	19%	0.680 / 0.02	0.0
E	21%	0.870 / 0.2	1.0
F	21%	1.28 / 0.17	1.0

Table 3.3: Scalability results of our ILP scheme over the IPC scheme

As can be seen from Table 3.3, the heuristic strategy used in our optimization scheme could provide very good solutions with very low optimally gap compared to running the solver without time limit and callback. Also, such a strategy provided substantial improvements over the IPC-driven algorithm while requiring a relatively low execution time overhead.

To further improve the scalability of our optimization scheme, we may relax the task allocation variables of the ILP model (Section 3.1.1) to get a linear programming (LP) model that can be solved more efficiently. For example, to solve the LP using the interiorpoint algorithm requires polynomial time [52] and the classical simplex algorithm typically requires polynomial time, on average, despite its exponential worst-case behavior. The approximate (fractional) solutions derived from LP relaxations of the ILP model would require additional tasks to be split/migrated among core types, but a bound on the maximum number of split tasks could be derived [6]. Finally, our ILP model is a variant of the classical generalized assignment problem (GAP) and several heuristic implementations described in the literature, for example in [72], could be adapted and used to solve the ILP in a more efficient way.

3.4 Linux implementation

In this section we describe our experience with an implementation on a real multi-core running Linux. Our optimization scheme runs on a 64-bit x86 quad-core system with Linux CentOS 6 and kernel version 2.6.32. We developed a user-level monitor module using the **perfmon2** library, as described in Section 3.2.1, to collect performance data of the threads and a module to execute the configuration algorithm. In our implementation, a monitor process with a sampling period of 200ms (the same as used in [86]) is attached to each thread running in the system. We empirically determined the time limit of 5ms to solve our ILP model that stops the Gurobi solver, even if no solution was found, and thread re-allocation period of 1000ms to migrate threads between different core types. These parameters were able to capture an acceptable trade-off between overhead and configuration quality/responsiveness in our experiments.

3.4.1 Emulated heterogeneous core system

In our implementation, we use a 64-bit x86 quad-core system (Figure 3.9) where one core runs at 3.2Ghz and the other three cores run at 0.8Ghz with a shared 6MB L3 (last level) cache and off-chip memory subsystem of 8GB. We consider that a large core is equivalent to three small cores (3:1 ratio) [59]. Each individual core has 64 KB of L1 Data/Instruction cache, and 512 KB of L2 cache. Unfortunately it was not possible to make any other changes to the cores to model core heterogeneity in a more precise way, since that would require access to restricted and proprietary tools [59].

Based on the system in use, we determine the capacity of the multi-core system with respect to the number of instructions each core type can execute, C_i , and number of memory requests that can be processed, B, as inputs to our ILP model.

Similarly to the simulation methodology described in Section 3.2, we derive the pro-



Figure 3.9: Block diagram of our target heterogeneous multi-core system

cessing capacity for each core type C_i by measuring the highest thread IPC of all the SPEC CPU benchmarks, running one thread at a time. We made sure no other threads were running in the same core and no other user-level threads were running in our quad core system. We then multiply the highest IPC with the respective core clock speed to figure out the instruction per second capacity for each core type. The IPC is the same for both core types because the cores have the same micro-architecture and only differ in clock speeds.

To determine the memory bandwidth B, we used the *LMbench* performance benchmark [75] that reports the memory capacity as the data movement rate (bytes per second) between the processor and memory subsystem. The value of B for our system is 156 million requests per second based on the measured memory capacity of ≈ 10 GB/s and cache lines of 64 bytes.

The details of our optimization solution are given in the configuration algorithm shown in Figure 3.10. Our configuration algorithm consists of four phases. The first phase reads monitored computational (IPS) and memory demands (LLC misses) of the running threads in the system. The second phase predicts and updates the thread demands on each core type, applying the performance prediction model described in Section 3.1.2. The third phase involves solving the ILP model (described in Section 3.1.1) with the updated input demands of all threads; and then storing the configuration solution sorted by increasing thread IPS.

The last phase of the configuration algorithm assigns the threads to available cores in the system, given the configuration solution obtained from threads-to-core types assignment. This thread allocation phase sorts the available cores of each type by increasing IPS load, and then assigns each thread to a respective core in a round robin fashion, starting with the least-loaded core from a list of available cores.

The configuration algorithm is invoked at every configuration period (1000ms) to read the updated monitoring values and, when necessary, apply the reassignment decisions of threads to cores in the system. To enforce such reassignment decisions, our scheme uses // 1. Read counters of threads running on their respective core: // computational (IPS) "c_mon" and memory (LLC misses) "b_mon" c_mon, b_mon <- perfmon() // 2. Apply prediction model and update computational "c" and // memory "b" demands for all threads on each core type c, b <- core_pred(c_mon, b_mon) // 3. Solve ILP model and store new configuration "config" is a vector // of tuples (i,k), where k is the thread to be allocated on core type i config <- bestConfig(c, b) Sort threads in config by decreasing order of IPS // 4. Assign threads to available cores, where "core_list(i)" is a list // of available cores of type i sorted by increasing IPS load for each (i, k) in config: Assign thread k to a core from core_list(i) in a round robin fashion

Figure 3.10: Configuration algorithm

Linux's sched_setaffinity system call. In this way the kernel's scheduler guarantees that a thread will run only on those cores specified by a given thread-to-core affinity.

3.4.2 Workload description and measurements

We compare the performance and energy consumption against Linux's scheduler using a mix of different workloads. Although it might seem unfair to compare our optimization to the standard Linux scheduler, because the latter was not designed for heterogeneous systems, this gives an idea of how much improvement can be achieved if today's OS schedulers, such as Linux CFS (Completely Fair Scheduler), were to be used on a heterogeneous system.

A random subset of the SPEC 2006 benchmark suite is used as the workload on the system as shown in Table 3.4. We run several different programs concurrently to occupy the cores and memory system that allows for evaluating the thread assignment decisions in a variety of situations. We adopt a similar workload configuration to the one described in [59], where each SPEC benchmark program is considered a thread in the system and the number of threads is equal to the sum of cores.

The workload keeps running on the system and the experiment ends when each thread has run at least once. While one or more threads are not finished, a thread can restart its execution as soon as it is finished. We take the average of the execution time of each thread that runs more than once. We determine the performance as the average of the

XX 7 1 - 1 1	
workload	Set of threads
R4-1	gobmk, cactusADM, mcf, GemsFDTD
R4-2	lbm , bwaves , bzip2 , namd
R4-3	gamess, bwaves, bzip2, GemsFDTD
R4-4	soplex, astar, 1bm, mcf
R4-5	bwaves, namd, GemsFDTD, tonto
R4-6	soplex, milc, astar, tonto
R4-7	soplex, milc, lbm, mcf
R4-8	astar, gobmk, mcf, namd
R4-9	soplex, calculix, bwaves, tonto
R4-10	gamess, milc, gobmk, cactusADM

Table 3.4: Workload composition of 4-thread combinations

execution times (wall clock) of all threads that have already ran in the multi-core system. We measure thread execution time using the *time* program in Linux.

We obtain the power consumption of the multi-core system directly using the *WattsUP Pro* meter with 1% accuracy [25]. To determine the energy consumption, the power measurements are read out and accumulated every second while the experiment is running. Note that the power measures represent the whole machine, not only the CPU, although the CPU consumes the major fraction of the total power. We multiply the sum of the execution time of all threads by the total measured energy consumption to derive the energy-delay product (EDP).

3.4.3 Performance gains and energy savings

Figure 3.11 shows the energy and performance improvements of our scheme compared with the Linux scheduler on the multi-core system. The comparison of a given workload is based on an instance of the performance of the Linux scheduler. As observed in [59], the Linux scheduler shows inherent performance variability because of arbitrary initial thread-to-core assignments.



Figure 3.11: Energy and performance improvements (%) of our scheme over Linux scheduler



Figure 3.12: Distribution of thread-to-core assignment for workload R4-2 — Best scenario



Figure 3.13: Distribution of thread-to-core assignment for workload R4-9 — Worst scenario

As shown in Figure 3.11, our scheme is able to improve the EDP of all workloads by an average of 15%. Some workloads have improvements of more than 35%, namely R4-2, R4-3, R4-10, where our scheme have correctly assigned threads to the available cores. However, our scheme performed worst than Linux scheduler for the workloads R4-8 and R4-9, with negative impact in the EDP metric. In Section 3.4.4 we will analyze the reasons why some workloads have performed well and others worse.

Considering the execution of all workloads, the average processing time required to solve the ILP model was 676 microseconds, with minimum, maximum and standard deviation of 590, 920 and 100 microseconds, respectively. Given the configuration period of 1000ms used for dynamic thread assignment, the overhead of solving the ILP model is consistently very low across all of the experiments; more precisely, 1ms/1000ms = 0.1%.

3.4.4 Best and worst case analysis

We first examine workload R4-2 where our approach provides the best improvement in both energy and performance. Figure 3.12 shows the fraction of time (%) that each thread

spends running on each core using the Linux scheduler (Figure 3.12(a)) and our scheme (Figure 3.12(b)). For example, by allowing **bwaves** to run about 64% of its lifetime on the large core, our scheme provides performance gains of 48% over the **bwaves** execution using the Linux scheduler, and an average of 30% for the overall R4-2. The energy delay in this workload is approximately 70%.

We also examine workload R4-9 where our approach performs the worst in terms of energy and performance. Looking at the distribution of thread-to-core assignment from Figure 3.13(a) we observe that Linux achieves good load balancing among all threads. As shown in Figure 3.13(b), our scheme assigned **soplex** to run on the large core for about 71% of its execution lifetime. However, the compute-intensive threads **calculix** and **tonto** were only allowed to run on the large core for about 1% and 6% of their execution lifetime, respectively.

Running our scheme with workload R4-9 introduced a particular unbalanced and "unfair" core assignment, resulting in large negative impact on the overall performance and energy optimization. In particular, soplex improves its performance by 26%, but impacts negatively both calculix and tonto by approximately 60%. We only have one large core and they both would benefit from running on this one. To address this phenomenon, we plan to extend our ILP model to spread threads more evenly across all cores and allow each thread to receive a fair share of the available core resources, proportionally to the thread's computational demands. As shown in our experiments, it is not beneficial that in some cases a thread gets a much larger share of a large core than other threads. We propose and evaluate in Chapter 4.3 a heuristic solution for this particular unfair phenomenon. We opted for a simple heuristic solution to highlight such a fairness aspect in the heterogeneous core system. We plan to incorporate in our ILP-based approach the insights revealed through developing the proportional-share heuristic (Chapter 4.3).

Our optimization scheme works by reallocating a thread to different cores when the performance profile of a thread changes, while running concurrently with other threads in the system. In our scheme, the number of LLC cache misses of a thread is used to measure its memory bandwidth requirements, that must be met to minimize resource contention and improve performance in the system. In addition to the inherent overheads of thread relocation in dynamic thread assignment schemes, severe performance degradation associated with co-running threads may be observed due to shared LLC among all threads and cores in the system. Even without exceeding memory bandwidth constraints, it is possible that LLC contention exist and negatively affect the performance of threads to very different degrees [70].

Currently, our optimization model does not explicitly address the issues of shared LLC when predicting the behavior of a thread on different core types. We collected the

performance data of each thread running individually on each core type to build our prediction model. Furthermore, accurate measurement of LLC shared events at user-level on Linux poses some practical limitations when multiple threads are running in the multicore system [14]. In particular, using **perfmon2** tool [27], we observed that a thread that is actively running on a core can inadvertently capture LLC events caused by activity of threads running on other cores. Depending on the memory-intensity of co-running threads (for example, when there is a thread with very high LLC miss rate), other threads can observe a conflation of their own LLC miss rates. That may help explain some of the poor decisions taken by our optimization approach in the real system.

3.5 Related work

There have been several prior research works that focused on exploring the benefits from heterogeneous multi-core systems. Most of the related work found in the literature strive to maximize the performance gain over applications running on these systems. Few related works explicitly deal with energy and performance efficiency metrics in their approach. The work described in [61] is pioneering in studying thread assignment policies considering different core types in order to optimize both energy and performance metrics. However, their work provides analysis of potential energy reduction of dynamic core assignments for each thread instead of determining a global optimized assignment for a set of threads to cores in the system.

A fine-grained power management technique for multi-core systems with homogeneous architecture but different fixed frequency/voltages [83] enables fast migration of threads between cores, showing energy gains comparable to systems having per-core dynamic volt-age/frequency scaling (DVFS). Their work also argues that per-core DVFS is expensive to be implemented in practice, meaning that heterogeneous multi-core systems are more likely to be adopted as an energy-efficient alternative in the near future. Similarly to our approach, previous works rely on hardware performance monitoring to determine the best matching of threads to cores [57, 92], although only focusing on performance gains. The monitoring framework outlined in [92] aims to provide software support to predict thread performance on heterogeneous multi-cores.

In the same direction, previous research have proposed local thread assignment decisions based on relative performance benefits of each thread running on different core types. For instance, to guide the choice of the best thread assignment, some works such as [61, 8] use a measure related to the thread IPC. In particular, the IPC-driven algorithm bases its decision on the thread IPC ratio between large and small core types. Similarly, the work described in [59] adopts a specific thread-to-core affinity measure, termed bias, which is related to the intensity of off-core (cache/memory) requests of each thread in the system. The metric used by bias scheduling can be determined by the number of off-core requests (LLC misses) per committed instructions. Both the IPC-driven algorithm and bias scheduling make local thread assignment decisions based on a relative gain metric with regard to running a thread on a particular core type. The intuition is that a thread with a high IPC ratio or low memory intensiveness bias ratio is expected to take best advantage of a large core rather than a small core. A comprehensive scheduler for heterogeneous core systems is proposed in [86] to maximize performance of both single-threaded and parallel workloads using the memory-intensity of threads to guide thread assignment decisions. In contrast to our work, they use only a single measure, due to the alleged correlation between IPC and LLC miss rate.

A scheduling technique for heterogeneous multi-core systems is described in [90]. However, their technique cannot cope with thread phase changes, nor does it scale well since the complexity of using off-line thread profiling becomes very high as the number of threads increases. Some research studies [62, 29, 35] analyze architectural characteristics of core types for optimizing the design area and power/performance efficiency in future multi-core systems. These works provide useful insights on enhanced capabilities of future heterogeneous multi-core systems but are orthogonal to the dynamic thread assignment problem. However, some works note that architectural or implementation limitations will hinder the efficiency of certain schemes. In particular, the proposed IPC-based scheme [8] needs to know the IPC of each core type before assigning threads to cores. In their implementation, the threads are executed for a small interval of time in each core, the IPC measured, and then the threads assigned. This incurs much overhead, as noticed in [90].

3.6 Summary

Heterogeneous multi-core systems present interesting challenges on thread assignment and scheduling research. Despite the challenging fact that core types have asymmetric computing capabilities, threads have time-varying real-time computational and memory requirements that need to be met accordingly. We propose an optimization scheme to effectively determine an energy efficient thread assignment based on an ILP model. Our scheme is able to dynamically change thread-to-core assignment aiming at meeting soft real-time performance and memory bandwidth constraints in heterogeneous multi-core systems.

According to simulation results, our scheme achieves large energy savings and performance gains for a variety of workloads and outperforms other proposed thread assignment schemes that do not address explicitly memory bandwidth constraints. For example, the improvement on energy-delay product over the state-of-the-art is about 20% to 40% depending on the workload. We also show a substantial decrease in tardiness. Additionally, we presented an implementation and experimental evaluation of our approach in a real heterogeneous multi-core system. Our approach successfully satisfied thread performance and memory bandwidth requirements for a variety of workloads composed of programs that concurrently lay emphasis on the cores and memory subsystem. We have shown energy-delay product gains of 15% (average) to 35% (maximum) for a variety of workloads compared to existing OS scheduler (Linux).
Chapter 4

Proportional share scheduling for heterogeneous multi-cores

As we have presented in Chapter 3, an optimized *energy-efficient* thread assignment has to match the thread demands with the capabilities of the heterogeneous cores. In this part of the work, we demonstrate that, in addition to satisfying the threads' computational demands, thread scheduling has to provide *fair allocation* of the platform resources to the threads. We develop a proportional-share scheduling strategy for heterogeneous multi-cores that leverages lottery/ticket mechanisms to provide *fairness* and optimize for combined performance and energy savings.

Our heuristic, *lucky scheduling*, relies on local thread reassignment decisions rather than solving an ILP optimization model. Lucky is a simple heuristic designed to demonstrate the essence of proportional-share scheduling decisions and highlight the benefits of providing fairness in big/small core allocation to the threads in the system. The running threads are given some number of tickets derived from runtime performance monitoring. The given tickets of a thread corresponds to its energy efficiency estimate and determine relative chance of all of the other threads competing for the available big cores. Experimental results show that lucky scheduling can provide better performance and energy savings over state-of-the-art heterogeneous-aware scheduling techniques.

4.1 Fairness in dynamic thread assignment

Recalling our discussion in Chapter 3, previous works have proposed taking advantage of performance counters provided by the hardware to determine the characteristics for the running threads and schedule them on the right core type [59, 87, 92]. To make thread scheduling decisions, these scheduling techniques calculate the thread *bias* through measuring either compute-intensity (IPC or IPS, instructions per cycle or per second) [8, 61] or memory intensity (e.g., LLC miss rate) [59, 86] which determines the thread execution efficiency between the core types. The intuition is that a thread with a high



Figure 4.1: Performance comparison between bias scheduling and big core fair sharing

IPS ratio or low-memory intensity bias is expected to take best advantage of a big core rather than a small core.

Most of those scheduling algorithms [8, 59, 61, 87, 90] are unfair by design since they assign to big cores the threads that experience the highest bias towards those cores. As such, some threads can monopolize the available big cores and hinder the progress of other threads. Figure 4.1 shows the results of an experiment where one memory-intensive and three compute-intensive threads were running on a quad-core multi-core system with one big core and three small cores (Section 4.2 describes our experimental setup). For this particular workload, we can observe a performance *speedup* of 1.47 times (energy consumption reduction of 37%) over running bias scheduling [59] by simply providing equally fair sharing (round-robin) of the big core among all threads (25% for each thread).

In particular, using bias scheduling in the experiment shown in Figure 4.1, the highest compute-intensive thread had possession of the big core 96% of the workload execution time, leading to severe performance degradation of the other compute-intensive threads. The workload execution time for each scheduling algorithm is the elapsed time of the last completed thread. The workload is long lived and threads are restarted until longest thread finishes.

In contrast to existing work, we make the case for proportional-share scheduling of threads in heterogeneous processor cores aimed at improving combined energy efficiency and performance. We show that allocating big/small cores proportionally to the threadto-core execution efficiency provides on average the best energy and performance gains.

4.2 Power/performance for heterogeneous cores

We consider a multi-core system having the following core types: big/fast cores targeted for high-performance and small/slow cores optimized for low-power. Such a system with

Core	Peak power	Idle power	Avg. power	Capacity
big	18.75 W	$9.625 { m W}$	$15.63 { m W}$	6,307 MIPS
small	$2.15 \mathrm{W}$	$0.7 \mathrm{W}$	1.6 W	1,592 MIPS

Table 4.1: Power and performance measures of heterogeneous cores

two distinct types of cores is able to capture most of the benefits from heterogeneity [20, 34]. As in Chapter 3, we use the following performance-asymmetric multi-core system in our experiments: a quad-core x86_64 chip capable of individual core frequency scaling where one core runs at 3.2Ghz and the other three cores run at 0.8Ghz. All cores share a L3 (last level) cache of 6MB and off-chip memory subsystem of 8GB.

Table 4.1 describes the power and performance characteristics of the heterogeneous cores in the system. The performance (capacity) of a core is defined as follows. We measured the highest thread IPC, by running the SPEC CPU benchmarks one thread at a time. Multiplying this number by the clock speed, we find out how many instructions can be executed in one second. We use MIPS (million instructions per second) as the measure of core computational capacity/performance.

Our multi-core system includes identical cores only differing in clock frequency. Recall that power consumption typically scales linearly with core frequency (Chapter 2). This means that power savings do not necessarily translate into energy savings, since reducing core power/frequency will extend thread execution time and increase energy consumption. Thus, we estimate core power consumption based on measurements [48] of a real heterogeneous platform configuration composed of typical big (Intel Xeon) and small (Intel Atom) cores [20].

In our multi-core system model, a big core delivers 4-fold performance but a small core is 2.2 times more power-efficient (i.e., MIPS per Watt). A small core consumes on average much less power than a big core and is also attractive in terms of the performance obtained in proportion to the power consumed (power-performance proportionality). This is because of its very low idle power (0.7W) and wide dynamic power range between idle and peak power (0.7W to 2.15W).

4.2.1 Thread performance/bias characterization

We use Linux 2.6.34 kernel with *perf* monitoring tool to gather hardware performance events namely retired_instructions and L3_cache_misses in order to characterize the behavior of a thread execution in the multi-core system. To allow isolation of individual LLC miss rate, we set up the Linux *perf* tool to read the 4E1 (L3 Cache Misses) raw register with different core selection masks, overcoming some monitoring issues we described in Chapter 3. We measure LLC misses per core since a LLC event is an event shared across all cores in the system [14].



Figure 4.2: Performance characterization of benchmarks programs while running alone on a big core

Each core runs a separate thread and thus we monitor the performance counters for a given core (thread) for a given monitoring interval. Given these counters, we compute the MIPS (million instructions per second) to determine the CPU demands of a thread. Similarly, we obtain the number of LLC misses per second, LLCMS, to characterize memory demands, where each LLC miss represents an off-chip memory request.

Existing thread assignment policies [8, 59, 86] work based on the correlation between the CPU load given by IPS (or MIPS, millions of IPS) or IPC (Instructions Per Cycle) and core stall-time indicators, such as LLC miss rate, which directly correlate to the amount of execution time in which the thread cannot retire instructions. However, we have observed that there is not always a clear one-to-one correlation between these two metrics in the execution of typical threads.

As shown in Figure 4.2, a single metric cannot clearly characterize some threads and schedule them to the right core type. For example, both MIPS and LLC misses can be increased for a given thread execution, such as the case of *milc* benchmark (64M LLC misses and 2K MIPS) when compared with *mcf* (18M LLC misses and 0.4K MIPS). Also, very similar MIPS can lead to very different memory intensity, such as between *lbm* (48M MPS, 2.4K MIPS) and *cactusADM* (8M MPS, 2.3K MIPS). In our work we explicitly use both performance measures aimed at augmenting thread assignment decision to meet thread performance demands.

We use real performance measurements when a thread is running on the same core type, and assume the next scheduling interval can be approximated by the current interval. We use the equation $MIPS_{j,k} = \alpha_j \cdot MIPS_{i,k} + \beta_j \cdot LLCMS_{i,k} + \gamma_j$ to estimate the performance behavior of a thread k currently running on a given core type i when it is assigned to a different core type j in the next scheduling interval. The linear regression model above is key to determine the energy efficiency $(MIPS^2/Watt)$ of a thread between big and small cores. The regression coefficients α_j , β_j and γ_j are derived from of-



Figure 4.3: Prediction error analysis of thread performance on different core types



Figure 4.4: Predicting thread MIPS from big to small core (bwaves) and small to big core (astar)

fline performance data collected running programs from the SPEC 2006 benchmark suite individually on each core type j. Running the method of least squares to the performance data yielded a coefficient correlation of approximately 98%.

Figure 4.3 shows the prediction error (given by the normalized root mean square deviation) for different benchmarks. We normalized the prediction errors by the range of measured values on the respective core type to make the results comparable. The average prediction error of less than 3% indicates that our estimation model can accurately predict the performance of a thread running on different core types. The highest prediction error was for the **astar** benchmark, considering the performance estimation from both small-to-big and big-to-small cores, 8% and 7%, respectively. The **bwaves** benchmark had highest prediction error (10.7%) for predicting big-to-small core.

Figure 4.4 illustrates the prediction of MIPS for the **bwaves** and **astar** benchmarks when running on a small core and intended to run a big core. The performance data were collected from a small core and used to predict the performance on a big core. The graph shows predicted versus actual values on a big core. We can observe that our prediction model can successfully capture the different phases of a thread in most cases. For practical purposes such inaccuracies in our prediction model did not prevent our scheduling strategy from achieving performance and energy improvements over existing thread scheduling techniques (Section 4.4).

4.3 Lucky scheduling

Lucky scheduling is designed to carry out thread assignment to heterogeneous cores for optimized performance and energy savings. The assignment of threads to cores are periodic (*reassignment interval*) to cope with thread execution phase changes. Lucky scheduling builds on the concepts and mechanism from the lottery scheduler [97] to implement such a dynamic scheduling strategy.

The novelty of lucky is in how the ticket assignment is done in the scheduling algorithm. Each thread receives a dynamic number of tickets which is determined by the *energy efficiency* ratio between running the thread on a big core vs small core.

4.3.1 Energy efficiency metric

The goal of the metric used by *lucky* scheduling to guide thread assignment decisions is to optimize for the *energy-delay product* (EDP) per instruction. In fact, we maximize the inverse of the energy-delay product, given by $MIPS^2/Watt$ [16], to facilitate the computation of the energy efficiency ratio between big and small cores (details presented in Section 4.3.2).

Figure 4.5 shows a comparison of the energy efficiency ratio between big and small cores when running the SPEC benchmarks on different core types. We can observe that the energy efficiency varies over different programs and can be used for guiding dynamic thread assignment decisions. Relying on previous experiments described in Chapter 3, we observe that similar behavior exists for different execution phases of each thread. That is, some threads go through compute-intensive or memory-intensive phases where the $MIPS^2/Watt$ changes during the course of their executions [61]. We use this information to periodically reassign the running threads to big or small cores in an energy-efficient manner.

Looking at Figure 4.5, we observe that the *astar* benchmark should run on a big core, whereas *zeusmp* is best suited on a small core to improve energy efficiency. Bias scheduling [59], for example, would map both programs on a small core type since both experience high memory stalls, about 14 million *LLCMS*. A typical compute-intensive thread has 1-2 million *LLCMS*. Recall Figure 4.2 that shows MIPS vs LLCMS measurements for those programs. We also noted that *bwaves* is an instance of benchmark that bias scheduling would run it on a small core because of its high memory-intensive (29 millions of *LLCMS*). However, *bwaves* is best suited for a big core to improve energy efficiency.



Figure 4.5: Energy-efficiency ratio (big-to-small core) of threads when running on different core types

The observations above indicate that explicitly taking into account the core power consumption in thread scheduling can improve energy-efficient scheduling decisions. Furthermore, different power/performance ratios can be observed in the design of heterogeneous multi-core systems. This makes it much more challenging to derive energy-efficient thread assignment decisions based only on either computational demands (MIPS) [8, 61] or memory stalls (LLC misses) [59, 86], individually.

4.3.2 Algorithm outline

We introduce the following notation for the heterogeneous multi-core system. Let N be the set of core types, N_i be the set of cores of type $i \in N$, and M be the set of all available cores; that is $M = N_1 \cup N_2 \cup \ldots N_n$, where n = |N|. Each core of type $i \in N$ has same computational capacity C_i (million instructions per second) and peak/busy power B_i and static/idle power I_i (Watts).

Let K be the set of threads to run in the system. Each thread $k \in K$ requires computational execution rate $MIPS_{i,k}$ and memory access rate $LLCMS_{i,k}$ when executing on core type $i \in N$. We estimate thread power consumption as $P_{i,k} = (B_i - I_i) \cdot (MIPS_{i,k} / C_i) + I_i$, when thread k runs on a core of type i. We estimate the energy consumption (in Joules) as $E = \sum_{k \in K} P_k \cdot S$, given the configuration of each thread allocated to a core type in a given scheduling interval S (in seconds). The energy efficiency of a thread-to-core assignment is $energy_efficiency(i, k) = MIPS_{i,k}^2 / P_{i,k}$ in a given scheduling interval.

In *lucky* scheduling, the ticket allocation of a thread $k \in K$ is determined periodically by its energy efficiency ratio between two types of cores: big and small cores (where $N = \{big, small\}$). A given thread is expected to run on a big vs small core proportionally to the number of tickets it holds. More precisely, allocating more tickets to a thread gives it a higher priority to run on a big core. This allows to adjust dynamically the priority of a given thread currently running on a small core to move to a big core, thereby reducing unfairness and improving overall performance in the system. The activity of always exchanging two running threads between different core types help preserve load balancing.

The initial thread assignment is given by the underlying operating system (OS) scheduler. Typically the OS assigns each thread to a core so that the workload is balanced across the available cores in the system. The algorithm of *lucky* scheduling is outlined as follows:

- 1. Measure $MIPS_{i,k}$ and $LLCMS_{i,k}$ of each thread $k \in K$ running on a core of type $i \in N$
- 2. **Predict** $MIPS_{j,k}$ on the other core of type $j \in N \{i\}$ (see Section 4.2.1)
- 3. Evaluate $big_core_benefit(k) = \frac{energy_efficiency(b,k)}{energy_efficiency(s,k)}$ for each thread k, big core b, and small core s
- 4. Generate a number of $tickets(k) = 100 \cdot big_core_benefit(k)$ to assign to each thread $k \in K$
- 5. Determine thread T that holds the winning_ticket given by a random number uniformly distributed between [0, total_tickets) where total_tickets = $\sum_{k \in K} tickets(k)$
- 6. Swap thread T with a thread T' in case T is not running on a big core, considering that T' has the minimum number of tickets and is running on the least-loaded big core.

We exploit a lottery-based approach to implement *lucky* scheduling because of the *simplicity* of its implementation including the *flexibility* when including/removing threads in the system. The current implementation of *lucky* scheduling uses a list data structure to keep threads updated with their current number of tickets and a global variable to track the sum of threads' tickets. The random number generator implemented in C++ standard library is used to select the winning ticket/thread.

Since the number of core types is small and fixed, the algorithm complexity is O(m) where m is the number of threads, because it is dominated by searching for a thread with *winning_ticket*. For future many-core systems with hundreds or thousands of cores, a more efficient implementation using balanced tree or heap structures could reduce the complexity to $O(\log m)$ [97].

4.4 Results

We evaluate the performance and energy consumption of **lucky** scheduling against the heterogeneous-aware **fair** share policy [87] and **bias** scheduling [59] using a mix of thread workloads. The fair share algorithm allocates the big core among threads in a round

robin fashion. For the bias scheduling, we compute $bias_k = LLCMS_{i,k} / MIPS_{i,k}$ for each thread k running on a core of type i. Threads with the lowest bias are mapped to the available big cores, whereas the highest bias are mapped to small cores.

Binding threads to cores of the same type is done via sched_setaffinity system call. The Linux scheduler thus makes scheduling decisions respecting this affinity. The reassignment interval is set up to 200ms which approximately corresponds to the Linux load balancing granularity, responsible for migrating threads among cores to maximize system utilization [65]. We found this interval suitable to capture thread phase changes while helping mitigate thread migration overhead and cold-cache effect.

4.4.1 Workload description

The SPEC 2006 benchmark suite was used as the workload on the system. Each SPEC benchmark program is considered a thread in the system and the number of threads is equal to the sum of cores [59]. Table 4.2 shows the different workloads tested in our experiments where each workload is a combination of four threads. We run several different benchmarks concurrently to create a mix of core and memory usage. The *ref* input option is used in all the SPEC benchmark programs. The benchmarks were selected to test a variety of CPU and memory requirements which are composed of 4-threads mixture varying from compute-intensive (4CI) to memory-intensive (4MI) workloads [86]. The workload 4P represent benchmarks that exhibit different phases and workloads 4R1, and 4R2 are random combinations of threads.

Workload	Set of programs
1CI-3MI	sjeng lbm milc soplex
2CI-2MI	bwaves tonto soplex mcf
3CI-1MI	povray sjeng bwaves soplex
4CI	calculix povray namd tonto
4MI	lbm milc mcf soplex
4P	astar bzip2 leslie3d milc
4R1	namd mcf astar bwaves
4R2	lbm bzip2 calculix GemsFDTD

Table 4.2: Workload composition of 4-thread combinations of the SPEC2006 benchmark

The experiment ends when each thread has run at least once; that is, until the longest thread finishes, also known as *makespan* in the literature. While the longest thread is not finished, the other threads restart their executions as soon as they are finished. We measure the execution time (wall clock) of a workload by calling Linux gettimeofday system call at the start and end of the workload execution. We estimate the energy consumption of the running threads in a workload using the formula from Section 4.3.2. We multiply the sum of estimated energy consumption of all threads executed in the



Figure 4.6: Improvements of lucky scheduling (%) over bias scheduling and big core fair policy

system by the workload execution time to derive the energy delay product (EDP).

4.4.2 Energy efficiency and performance gains

Figure 4.6 shows an experiment to demonstrate that inherently unfair thread scheduling leads to more performance loss than improvement for the given mixed set of workloads. When considering all workloads executions, a simple big core **fair** policy provides EDP gains of 16% over **bias** scheduling. Beyond that, **lucky** scheduling outperforms big core fair policy in EDP by 12% (avg.) and 20% (max.). This indicates that energy-efficient proportional-share brings higher performance/energy improvements rather than simply providing equally fair sharing of the big core among all threads. Lucky scheduling achieved better EDP when compared to bias scheduling over all workloads executions (avg. 39% and max. 51%).

The big core fair policy outperformed lucky scheduling in EDP by 4% in the workload 3CI-1MI and 12% in the workload 4MI. This particularly highlights that more biased/unfair decisions can negatively affect the energy and performance for some workloads. For example, for workload 4MI, lucky scheduling allocated a specific memoryintensive thread (mcf) to the big core for about 9%, in contrast to 25% using the fair policy. As shown in Figure 4.7, the benchmark mcf has high MIPS phases, when it is suitable to run the big core, which last around 25% of the time (as provided by fair policy) rather than 9% (as provided by lucky).

Also, we can observe that the other memory-intensity programs (soplex, milc, lbm) in the composition of workload 4MI have higher MIPS than mcf (see Figure 4.2) and, accordingly, these programs obtained more tickets/bias to run on the big core than the mcf program. Clearly, this particular big core distribution hurts mcf performance, although it is, on average, small core biased. This highlights the bias/fairness trade-off decisions.

Regarding the execution of workload 4P, lucky outperformed big core fair (by 19%) and bias (by 51%) policies. Figure 4.8 shows the allocation distribution of the big core



Figure 4.7: MIPS over time for mcf running on a big core



Figure 4.8: Big core distribution of workload 4P with fair, bias and lucky scheduling

among the threads in the system. We can observe that bias scheduling assigned bzip2 to run on the big core for about 87% of the time, monopolizing the big core. We can see that lucky scheduling was able to provide adequate balance between fairness and core execution bias/efficiency in this particular workload, and most cases in Figure 4.6.

We demonstrate that providing *fair-share* in addition to *bias* scheduling is a very important aspect that should be taken into account when matching the threads' computational demands to the multi-core resources over time. In future experiments, we may exploit *ticket inflation* to allocate additional tickets to the threads that are already running on a big core, giving more chances to keep those threads running on the same type of core. This can help minimize migration overhead and preserve cache affinity. Sensitivity analysis is needed to determine the best parameter for the ticket inflation and which workloads would benefit from such a thread scheduling adjustment.

Migrating specific threads between cores with different processing capabilities can also cause performance variability due to contention in the shared resources (last-level cache, memory controller/bus). Extending lucky scheduling to incorporate explicitly (1) cache/memory contention awareness [71, 50] and (2) real-time/tardiness/laxity aspects of tasks in the ticket estimate/assignment are interesting avenues for future work. In particular, real-time performance guarantees are crucial for latency-sensitive cloud applications such as web search and media streaming [30].

4.5 Related work

Evolving beyond traditional homogeneous systems, research works have shown that improved performance and energy-efficiency benefits can be achieved by adopting heterogeneous multi-core systems [61, 29]. To determine the thread execution efficiency between the core types, state-of-the-art scheduling techniques calculate the thread *bias* through measuring either compute-intensity (IPC or IPS, instructions per cycle or per second) [8, 61] or memory intensity (e.g., LLC miss rate) [59, 86].

A typical unfair scheduler such as bias scheduling [59, 86] may not be appropriate in situations when more than one compute-intensive thread can benefit from running on a big core. On the other hand, existing fair scheduling algorithms aim to share the available big cores in the *same degree* among all threads in the system [67, 87]. Also, current scheduling algorithms strive to maximize the performance gain over threads running on the heterogeneous cores. They do not take into account the actual core power consumption to make scheduling decisions and may deliver less energy-efficient thread assignments.

4.6 Summary

Energy-efficient heterogeneous multi-core systems pose challenging scheduling problems. We advocated a proportional-share scheduling strategy based on lottery/ticket mechanisms that optimizes for combined performance and energy savings. We proposed a simple and effective way to determine ticket/thread assignment by estimating thread performance and energy efficiency between core types in the system emulated using different clock speeds. We demonstrated that inherently unfair thread scheduling may cause, on average, more energy/performance loss than improvements for the given set of workloads. Lucky scheduling has shown energy savings and performance improvements over state-ofthe-art thread assignment schemes designed for heterogeneous multi-core systems.

Chapter 5

Conclusion and future directions

In this dissertation, we described an approach based on optimization models and heuristic techniques to perform dynamic optimized assignment of tasks to processors in server systems, while meeting power and performance requirements. We tailored our optimization approach to address the cases of heterogeneous virtualized clusters and multi-core systems, developing specialized management strategy and implementation for each case.

In Chapter 2, we developed and demonstrated the feasibility of our optimization approach for power and performance management in virtualized servers. We conducted experiments with realistic Web traffic driven by time-varying demands with real system measurements. Such an optimization strategy achieved energy savings of 14% compared to DVFS-enabled cluster and 52% compared to typical uncontrolled system (used as baseline), while simultaneously meeting applications' performance goals as specified in their response time deadlines.

In Chapter 3, we introduced timeliness as a first-class concern in scheduling of heterogeneous cores and solved an optimization problem with constraints on bandwidth to the memory and core utilizations. Our approach successfully satisfied thread performance and memory bandwidth requirements, showing improvement on energy-delay product over the state-of-the-art of 20% to 40% depending on the workload. We also show a substantial decrease in tardiness, which measured how well the system did in completing the task execution on a timely fashion. In comparison with a Linux task scheduler implementation, we show an average energy-delay product gains of 15%, with average energy savings of 11% and performance improvements of 7%.

In Chapter 4, we devised a lottery scheduling scheme, lucky, in which the number of tickets allocated to a thread is determined at runtime by its energy efficiency ratio on big and small cores. This strategy was used to assign to each thread the appropriate chance to execute in the most suited core type. In this way, lucky scheduling was able to provide an adequate balance between bias and fairness on using big/small cores. Lucky scheduling outperformed a big core fair (round-robin) policy in EDP by 12% (avg.) and

20% (max.), and achieved better EDP when compared to state-of-the-art bias scheduling over all workloads executions (avg. 39% and max. 51%).

We have identified the following research directions:

- A promising continuation of our work would be to develop a *comprehensive approach* that optimizes for both heterogeneous virtualized servers and multi-cores processors. This would require further investigation on designing an optimization approach as multiple interacting feedback/management loops, possibly in a hierarchical manner.
- To achieve energy-efficient program execution in *multi-threaded benchmarks*, one direction is to speed up the thread(s) that are (likely) to be on the critical path by running them on the big/high-performance core(s) and allow the other threads to execute in parallel on the small/low-power cores [93]. One challenging aspect will be to determine whether or not threads are on the program critical path; for example, based on correlation of run-time performance data or program code anno-tation/instrumentation [49].
- Exploring thread consolidation may provide additional energy-savings by maximizing the utilization of a set of active cores and quickly powering down idle cores, bringing them up when thread workloads increase [13]. However, switching cores on/off indiscriminately can consume extra time and power/energy that should be taken into account. This will require the characterization of performance degradation/overhead when co-running workload threads with shared resources (cache/memory) [15, 69] and powering up/down the cores in the system.

APPENDIX A – Publications

- Vinicius Petrucci, Orlando Loques, Daniel Mossé. Lucky scheduling for energyefficient heterogeneous multicore systems. The 2012 Workshop on Power-Aware Computing and Systems (HotPower'12), co-located with the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12), Hollywood, CA. — Workshop at Conference CAPES/Qualis A1.
- Luca Lugini, Vinicius Petrucci and Daniel Mossé. Online Thread Assignment for Heterogeneous Multicore Systems. The 2012 International Workshop on Embedded Multicore Systems (ICPP- EMS 2012), Pittsburgh, PA. — Workshop at Conference CAPES/Qualis A2.
- Vinicius Petrucci, Orlando Loques, Daniel Mossé, Rami Melhem, Neven Gazala, Sameh Gobriel. Thread assignment optimization with real-time performance and memory bandwidth guarantees for energy-efficient heterogeneous multi-core systems. The 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12), Beijing, China, April, 2012. — Conference CAPES/Qualis A2.
- 4. Renaud Masson, Thibaut Vidal, Julien Michallet, Puca Huachi Vaz Penna, Vinicius Petrucci, Anand Subramanian, Hugues Dubedout. A Hybrid Large Neighborhood and Local Search for the Machine Reassignment Problem. 25th European Conference on Operations Research (EURO), Vilnius, Lithuania, 2012.
- Hugo Kramer, Vinicius Petrucci, Anand Subramanian, Eduardo Uchoa. A column generation approach for power-aware optimization of virtualized heterogeneous server clusters. Computers & Industrial Engineering 63(3): 652-662 (2012) — Journal CAPES/Qualis A2.
- Vinicius Petrucci, Enrique V. Carrera, Orlando Loques, Julius Leite, Daniel Mossé. Optimized Management of Power and Performance for Virtualized Heterogeneous Server Clusters. 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid (CCGrid'11), Newport Beach, CA, USA, 2011. — Conference CAPES/Qualis A1.

- Carlos Oliveira, Vinicius Petrucci, Orlando Loques. Using Virtual Machine Replication for Dynamic Configuration of Multi-tier Internet Services. IADIS International Conference WWW-Internet, Rio de Janeiro, RJ, Brazil, 2011.
- Carlos Oliveira, Vinicius Petrucci, Orlando Loques. Impact of server dynamic allocation on the response time for energy-efficient virtualized web clusters. 12th Brazillian Workshop on Real-Time and Embedded Systems (WTR), Gramado-RS, Brazil, 2010.
- Vinicius Petrucci, Orlando Loques, Daniel Mossé. A dynamic optimization model for power and performance management of virtualized clusters. 1st International Conference on Energy-Efficient Computing and Networking. In cooperation with ACM SIGCOMM. University of Passau, Germany, 2010.
- 10. Vinicius Petrucci, Orlando Loques, Daniel Mossé. Dynamic optimization of power and performance for virtualized server clusters. 25th ACM SAC (Short Paper/Poster on Power-Aware Design and Optimization Track), Sierre, Switzerland, 2010.
- Vinicius Petrucci, Orlando Loques, Daniel Mossé. Dynamic Configuration Support for Power-Aware Virtualized Server Clusters. 21th Euromicro Conference on Real-Time Systems (WiP Session), Dublin, Ireland, 2009.
- Vinicius Petrucci, Orlando Loques, Daniel Mossé. A Dynamic Configuration Model for Power-Efficient Virtualized Server Clusters. 11th Brazillian Workshop on Real-Time and Embedded Systems (WTR), Recife-PE, Brazil, 2009.

Bibliography

- ANDERSEN, D. G.; FRANKLIN, J.; KAMINSKY, M.; PHANISHAYEE, A.; TAN, L.; VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM* SIGOPS 22nd symposium on Operating systems principles (SOSP'09) (New York, NY, USA, 2009), ACM, pp. 1–14.
- [2] ARLITT, M.; JIN, T. Workload characterization of the 1998 world cup web site. Tech. rep., IEEE Network, 1999.
- [3] BARHAM, P.; DRAGOVIC, B.; FRASER, K.; HAND, S.; HARRIS, T.; HO, A.; NEUGEBAUER, R.; PRATT, I.; WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.
- [4] BARROSO, L. A. The price of performance. ACM Queue 3, 7 (2005), 48–53.
- [5] BARROSO, L. A.; HÖLZLE, U. The case for energy-proportional computing. Computer 40, 12 (2007), 33–37.
- [6] BARUAH, S. Task partitioning upon heterogeneous multiprocessor platforms. In IEEE Real-Time Systems and Embedded Technology and Applications Symposium (2004), pp. 536–543.
- BARYSHNIKOV, Y.; COFFMAN, E.; PIERRE, G.; RUBENSTEIN, D.; SQUILLANTE, M.; YIMWADSANA, T. Predictability of web-server traffic congestion. In 10th International Workshop on Web Content Caching and Distribution (WCW) (Sept. 2005), pp. 97–103.
- [8] BECCHI, M.; CROWLEY, P. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Computing frontiers* (2006).
- [9] BERTINI, L.; LEITE, J. C. B.; MOSSÉ, D. Power optimization for dynamic configuration in heterogeneous web server clusters. *Journal of Systems and Software 83*, 4 (2010), 585 – 598.
- [10] BERTOSSI, A. A.; MANCINI, L. V. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems* 7 (1994), 229–245.

- [11] BIANCHINI, R.; RAJAMONY, R. Power and energy management for server systems. Computer 37, 11 (2004), 68–74.
- [12] BICHLER, M.; SETZER, T.; SPEITKAMP, B. Capacity planning for virtualized servers. Workshop on Information Technologies and Systems (WITS), Milwaukee, Wisconsin, USA (2006).
- [13] BILGIR, O.; MARTONOSI, M.; ; WU, Q. Exploring the potential of cmp core count management on data center energy savings. In *In Proceedings of the 3rd Workshop* on Energy Efficient Design (WEED) (2011).
- [14] BLAGODUROV, S.; FEDOROVA, A. User-level scheduling on numa multicore systems under linux. In *Linux Symposium* (2011).
- [15] BLAGODUROV, S.; ZHURAVLEV, S.; FEDOROVA, A. Contention-aware scheduling on multicore systems. ACM Trans. Comput. Syst. 28 (December 2010), 8:1–8:45.
- BROOKS, D. M.; BOSE, P.; SCHUSTER, S. E.; JACOBSON, H.; KUDVA, P. N.; BUYUKTOSUNOGLU, A.; WELLMAN, J.-D.; ZYUBAN, V.; GUPTA, M.; COOK, P. W. Power-aware microarchitecture: Design and modeling challenges for nextgeneration microprocessors. *IEEE Micro 20*, 6 (Nov. 2000), 26–44.
- [17] CHASE, J. S.; ANDERSON, D. C.; THAKAR, P. N.; VAHDAT, A. M.; DOYLE, R. P. Managing energy and server resources in hosting centers. SIGOPS Oper. Syst. Rev. 35, 5 (2001), 103–116.
- [18] CHENG, B. H.; GIESE, H.; INVERARDI, P.; MAGEE, J.; DE LEMOS, R. 08031
 software engineering for self-adaptive systems: A research road map. In Software
 Engineering for Self-Adaptive Systems (Dagstuhl, Germany, 2008), B. H. C. Cheng,
 R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., no. 08031 in Dagstuhl
 Seminar Proceedings, Schloss Dagstuhl Leibniz-Zentrum fuer Informatik, Germany.
- [19] CHENG, S.-W. Rainbow: cost-effective software architecture-based self-adaptation. PhD thesis, Pittsburgh, PA, USA, 2008. AAI3305807.
- [20] CHITLUR, N.; SRINIVASA, G.; HAHN, S.; GUPTA, P. K.; REDDY, D.; KOUFATY, D.; BRETT, P.; PRABHAKARAN, A.; ZHAO, L.; IJIH, N.; SUBHASCHANDRA, S.; GROVER, S.; JIANG, X.; IYER, R. Quickia: Exploring heterogeneous architectures on real prototypes. In *IEEE 18th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2012), HPCA '12, IEEE Computer Society, pp. 1–8.
- [21] CHURCH, K.; GREENBERG, A.; HAMILTON, J. On delivering embarrassingly distributed cloud services. In *HotNets* (2008).

- [22] DANTZIG, G. B.; WOLFE, P. Decomposition principle for linear programs. Operations research 8, 1 (1960), 101–111.
- [23] DEVI, U. C.; ANDERSON, J. H. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Syst. 38* (February 2008), 133–189.
- [24] DINDA, P. A.; O'HALLARON, D. R. Host load prediction using linear models. Cluster Computing 3, 4 (2000), 265–280.
- [25] ELECTRONIC EDUCATIONAL DEVICES. Watts Up PRO. http://www.wattsupmeters.com/, 2010.
- [26] ELNOZAHY, E. N.; KISTLER, M.; RAJAMONY, R. Energy-efficient server clusters. In Proceedings of the 2nd international conference on Power-aware computer systems (Berlin, Heidelberg, 2003), PACS'02, Springer-Verlag, pp. 179–197.
- [27] ERANIAN, S. Perfmon2: a flexible performance monitoring interface for linux. In Proceedings of the Linux Symposium (2006), pp. 269–287.
- [28] FAN, X.; WEBER, W.-D.; BARROSO, L. A. Power provisioning for a warehousesized computer. In ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture (New York, NY, USA, 2007), ACM, pp. 13–23.
- [29] FEDOROVA, A.; SAEZ, J. C.; SHELEPOV, D.; PRIETO, M. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM 52* (December 2009).
- [30] FERDMAN, M.; ADILEH, A.; KOCBERBER, O.; VOLOS, S.; ALISAFAEE, M.; JEVDJIC, D.; KAYNAK, C.; POPESCU, A. D.; AILAMAKI, A.; FALSAFI, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2012), ASP-LOS '12, ACM, pp. 37–48.
- [31] FILANI, D.; HE, J.; GAO, S.; RAJAPPA, M.; KUMAR, A.; SHAH, P.; NA-GAPPAN, R. Dynamic data center power management: Trends, issues, and solutions. Intel Technology Journal. http://www.intel.com/technology/itj/2008/v12i1/6-datacenter/1-abstract.htm (February 2008), 2008.
- [32] FRIESEN, D. K.; LANGSTON, M. A. Variable sized bin packing. SIAM J. Comput. 15, 1 (Feb. 1986), 222–230.
- [33] GANDHI, A.; HARCHOL-BALTER, M.; DAS, R.; LEFURGY, C. Optimal power allocation in server farms. In *Proceedings of the eleventh international joint conference*

on Measurement and modeling of computer systems (New York, NY, USA, 2009), SIGMETRICS '09, ACM, pp. 157–168.

- [34] GREENHALGH, P. Big.LITTLE processing with ARM CortexTM-A15 and Cortex-A7. White Paper, 2011.
- [35] GUPTA, V.; KNAUERHASE, R.; SCHWAN, K. Attaining system performance points: revisiting the end-to-end argument in system design for heterogeneous many-core systems. SIGOPS Oper. Syst. Rev. 45 (February 2011).
- [36] GUROBI OPTIMIZATION INC. Gurobi optimizer version 4.5. http://www.gurobi.com/, 2011.
- [37] HALETKY, E. L. VMware ESX Server in the Enterprise: Planning and Securing Virtualization Servers.
- [38] HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B.; REUTEMANN, P.; WIT-TEN, I. H. The weka data mining software: an update. SIGKDD Explor. Newsl. 11 (November 2009), 10–18.
- [39] HAMILTON, J. Energy proportional datacenter networks. http://perspectives. mvdirona.com/2010/08/01/EnergyProportionalDatacenterNetworks.aspx, 2010.
- [40] HAYES, B. Cloud computing. Commun. ACM 51, 7 (2008), 9–11.
- [41] HEATH, T.; DINIZ, B.; CARRERA, E. V.; MEIRA, JR., W.; BIANCHINI, R. Energy conservation in heterogeneous server clusters. In ACM SIGPLAN symposium on Principles and practice of parallel programming (New York, NY, USA, 2005), PPoPP '05, ACM, pp. 186–195.
- [42] HENNESSY, J. L.; PATTERSON, D. A. Computer Architecture, Fourth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
- [43] HOELZLE, U.; BARROSO, L. A. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 1st ed. Morgan and Claypool Publishers, 2009.
- [44] HORVATH, T.; ABDELZAHER, T.; SKADRON, K.; LIU, X. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Transactions on Computers 56*, 4 (2007), 444–458.
- [45] ILOG, INC. CPLEX, 2009. http://www.ilog.com/products/cplex/.

- [46] INTEL CORP. Intel processor specifications. http://ark.intel.com/, 2011.
- [47] JALEEL, A. Memory characterization of workloads using instrumentation-driven simulation. http://www.glue.umd.edu/ ajaleel/workload/, 2011.
- [48] JANAPA REDDI, V.; LEE, B. C.; CHILIMBI, T.; VAID, K. Web search using mobile cores: quantifying and mitigating the price of efficiency. In 37th annual International Symposium on Computer Architecture (New York, NY, USA, 2010), ISCA '10, ACM, pp. 314–325.
- [49] JOAO, J. A.; SULEMAN, M. A.; MUTLU, O.; PATT, Y. N. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the seventeenth* international conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2012), ASPLOS '12, ACM, pp. 223–234.
- [50] KAMBADUR, M.; MOSELEY, T.; HANK, R.; KIM, M. A. Measuring interference between live datacenter applications. In *Proceedings of the International Conference* on High Performance Computing, Networking, Storage and Analysis (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 51:1–51:12.
- [51] KANDASAMY, N.; ABDELWAHED, S.; HAYES, J. Self-optimization in computer systems via on-line control: Application to power management. In *International Conference on Autonomic Computing* (2004), pp. 54–61.
- [52] KARMARKAR, N. A new polynomial-time algorithm for linear programming. In Proceedings of the sixteenth annual ACM symposium on Theory of computing (New York, NY, USA, 1984), STOC '84, ACM, pp. 302–311.
- [53] KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. Computer 36, 1 (Jan. 2003), 41–50.
- [54] KHANNA, G.; BEATY, K.; KAR, G.; KOCHUT, A. Application performance management in virtualized server environments. 10th IEEE/IFIP Network Operations and Management Symposium (0-0 2006), 373–381.
- [55] KHARGHARIA, B.; HARIRI, S.; YOUSIF, M. S. Autonomic power and performance management for computing systems. *Cluster Computing* 11, 2 (2008), 167–181.
- [56] KIM, W.; GUPTA, M. S.; WEI, G.-Y.; BROOKS, D. System level analysis of fast, per-core dvfs using on-chip switching regulators. In 14th International Conference on High-Performance Computer Architecture (HPCA-14 2008), 16-20 February 2008, Salt Lake City, UT, USA (2008), IEEE Computer Society, pp. 123–134.

- [57] KNAUERHASE, R.; BRETT, P.; HOHLT, B.; LI, T.; HAHN, S. Using OS observations to improve performance in multicore systems. *IEEE Micro 28* (May 2008).
- [58] KOOMEY, J. G.; BERARD, S.; SANCHEZ, M.; WONG, H. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing* 33 (2011), 46–54.
- [59] KOUFATY, D.; REDDY, D.; HAHN, S. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys'10*.
- [60] KRAMER, H. H.; PETRUCCI, V.; SUBRAMANIAN, A.; UCHOA, E. A column generation approach for power-aware optimization of virtualized heterogeneous server clusters. *Computers & Industrial Engineering 63*, 3 (2012), 652 – 662.
- [61] KUMAR, R.; FARKAS, K. I.; JOUPPI, N. P.; RANGANATHAN, P.; TULLSEN, D. M. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36* (2003).
- [62] KUMAR, R.; TULLSEN, D. M.; JOUPPI, N. P. Core architecture optimization for heterogeneous chip multiprocessors. In PACT (2006), pp. 23–32.
- [63] KUSIC, D.; KEPHART, J. O.; HANSON, J. E.; KANDASAMY, N.; JIANG, G. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing* 12, 1 (2009), 1–15.
- [64] LE, K.; BIANCHINI, R.; MARTONOSI, M.; NGUYEN, T. Cost-and Energy-Aware Load Distribution Across Data Centers, 2009.
- [65] LE, T. M. A study on linux kernel scheduler version 2.6.32. Available online: http: //www.scribd.com/thangmle/d/24111564-Project-Linux-Scheduler-2-6-32.
- [66] LE SUEUR, E.; HEISER, G. Dynamic voltage and frequency scaling: the laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems* (Berkeley, CA, USA, 2010), HotPower'10, USENIX Association, pp. 1–8.
- [67] LI, T.; BRETT, P.; KNAUERHASE, R.; KOUFATY, D.; REDDY, D.; HAHN, S. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *HPCA'10*.
- [68] LIU, H. A measurement study of server utilization in public clouds. In 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC) (Dec. 2011), pp. 435–442.

- [69] MARS, J.; TANG, L.; HUNDT, R.; SKADRON, K.; SOFFA, M. L. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2011), MICRO-44 '11, ACM, pp. 248–259.
- [70] MARS, J.; TANG, L.; SOFFA, M. L. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (New York, NY, USA, 2011), HiPEAC '11, ACM, pp. 167–176.
- [71] MARS, J.; VACHHARAJANI, N.; HUNDT, R.; SOFFA, M. L. Contention aware execution: online contention detection and response. In *CGO '10*.
- [72] MARTELLO, S.; TOTH, P. Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc., New York, NY, USA, 1990.
- [73] MASSON, R.; VIDAL, T.; MICHALLET, J.; PENNA, P. H. V.; PETRUCCI, V.; SUBRAMANIAN, A.; DUBEDOUT, H. An iterated local search heuristic for multicapacity bin packing and machine reassignment problems. *Tech. Rep., CIRRELT-*2012-70 (Submitted for publication) (2012).
- [74] MCKINSEY & COMPANY. Revolutionizing data center efficiency. http://uptimeinstitute.org, 2008.
- [75] MCVOY, L. W.; STAELIN, C. Imbench: Portable tools for performance analysis. In USENIX Annual Technical Conference (1996), pp. 279–294.
- [76] MEISNER, D.; GOLD, B. T.; WENISCH, T. F. Powernap: eliminating server idle power. In Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (New York, NY, USA, 2009), AS-PLOS '09, ACM, pp. 205–216.
- [77] MOSBERGER, D.; JIN, T. httperf a tool for measuring web server performance. SIGMETRICS Perform. Eval. Rev. 26, 3 (1998), 31–37.
- [78] NEW YORK TIMES. Power, pollution and the internet. http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vastamounts-of-energy-belying-industry-image.html, 2012.
- [79] OPENNEBULA. The open source toolkit for cloud computing. http://opennebula.org/, 2010.

- [80] PETRUCCI, V.; LOQUES, O.; MOSSÉ, D. Dynamic configuration support for poweraware virtualized server clusters. In WIP Session of the 21th Euromicro Conference on Real-Time Systems (2009).
- [81] PETRUCCI, V.; LOQUES, O.; MOSSÉ, D. Dynamic optimization of power and performance for virtualized server clusters. In SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing (New York, NY, USA, 2010), ACM, pp. 263–264.
- [82] QURESHI, A.; WEBER, R.; BALAKRISHNAN, H.; GUTTAG, J.; MAGGS, B. Cutting the Electric Bill for Internet-Scale Systems. In ACM SIGCOMM (Barcelona, Spain, August 2009).
- [83] RANGAN, K. K.; WEI, G.-Y.; BROOKS, D. Thread motion: fine-grained power management for multi-core systems. In *ISCA* (2009), pp. 302–313.
- [84] RANGANATHAN, P. Recipe for efficiency: principles of power-aware computing. Commun. ACM 53, 4 (2010), 60–67.
- [85] RUSU, C.; FERREIRA, A.; SCORDINO, C.; WATSON, A. Energy-efficient realtime heterogeneous server clusters. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium* (Washington, DC, USA, 2006), RTAS '06, IEEE Computer Society, pp. 418–428.
- [86] SAEZ, J. C.; PRIETO, M.; FEDOROVA, A.; BLAGODUROV, S. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys'10*.
- [87] SAEZ, J. C.; SHELEPOV, D.; FEDOROVA, A.; PRIETO, M. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. J. Parallel Distrib. Comput. 71, 1 (Jan. 2011), 114–131.
- [88] SANTANA, C.; LEITE, J.; MOSSE, D. Load forecasting applied to soft real-time web clusters. In ACM SAC (Sierre, Switzerland, March 2010).
- [89] SHARMA, V.; THOMAS, A.; ABDELZAHER, T.; SKADRON, K.; LU, Z. Poweraware qos management in web servers. In *The 24th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2003), RTSS '03, IEEE Computer Society, pp. 63–.
- [90] SHELEPOV, D.; SAEZ ALCAIDE, J. C.; JEFFERY, S.; FEDOROVA, A.; PEREZ, N.; HUANG, Z. F.; BLAGODUROV, S.; KUMAR, V. HASS: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.* 43 (April 2009).

- [91] SRIKANTAIAH, S.; KANSAL, A.; ZHAO, F. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and* systems (Berkeley, CA, USA, 2008), HotPower'08, USENIX Association, pp. 10–10.
- [92] SRINIVASAN, S.; ZHAO, L.; ILLIKKAL, R.; IYER, R. Efficient interaction between os and architecture in heterogeneous platforms. *SIGOPS Oper. Syst. Rev.* 45 (February 2011).
- [93] SULEMAN, M. A.; MUTLU, O.; QURESHI, M. K.; PATT, Y. N. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of* the 14th international conference on Architectural support for programming languages and operating systems (New York, NY, USA, 2009), ASPLOS '09, ACM, pp. 253–264.
- [94] THE APACHE SOFTWARE FOUNDATION. Apache HTTP server version 2.2. http://httpd.apache.org/docs/2.2/, 2008.
- [95] VERMA, A.; AHUJA, P.; NEOGI, A. pMapper: Power and migration cost aware application placement in virtualized systems. In *Middleware'08* (2008), pp. 243–264.
- [96] VIDAL, T.; DUBEDOUT, H.; MASSON, R.; MICHALLET, J.; PENNA, P.; PETRUCCI, V.; SUBRAMANIAN, A. A hybrid large neighborhood and local search for the machine reassignment problem. In 25th European Conference on Operations Research (2012).
- [97] WALDSPURGER, C. A.; WEIHL, W. E. Lottery scheduling: flexible proportionalshare resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1994), OSDI '94, USENIX Association.
- [98] WANG, Y.; WANG, X.; CHEN, M.; ZHU, X. Power-efficient response time guarantees for virtualized enterprise servers. In *Proceedings of the 2008 Real-Time Systems Symposium* (Washington, DC, USA, 2008), RTSS '08, IEEE Computer Society, pp. 303–312.