UNIVERSITAT POLITÈCNICA DE CATALUNYA

Exploring Coordinated Software and Hardware Support for Hardware Resource Allocation

by

Carlos Santieri de Figueiredo Boneti

A thesis submitted in partial fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

in the
Department of Computer Architecture

2009

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Exploring Coordinated Software and Hardware Support for Hardware Resource Allocation

by

Carlos Santieri de Figueiredo Boneti

Advisors: Francisco J. Cazorla
BARCELONA SUPERCOMPUTING CENTER
Roberto Gioiosa
IBM TJ. WATSON RESEARCH CENTER
Mateo Valero Cortés
UNIVERSITAT POLITÈCNICA DE CATALUNYA

A thesis submitted in partial fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

in the
Department of Computer Architecture

2009

“An intellectual is someone whose mind watches itself.”

Albert Camus

“The difference between the right word and the almost right word is the difference between lightning and the lightning bug.”

Mark Twain

Abstract

Multithreaded processors are now common in the industry as they offer high performance at a low cost. Traditionally, in such processors, the assignation of hardware resources between the multiple threads is done implicitly, by the hardware policies. However, a new class of multithreaded hardware allows the explicit allocation of resources to be controlled or biased by the software. Currently, there is little or no coordination between the allocation of resources done by the hardware and the prioritization of tasks done by the software.

This thesis targets to narrow the gap between the software and the hardware, with respect to the hardware resource allocation, by proposing a new explicit resource allocation hardware mechanism and novel schedulers that use the currently available hardware resource allocation mechanisms.

It approaches the problem in two different types of computing systems: on the high performance computing domain, we characterize the first processor to present a mechanism that allows the software to bias the allocation hardware resources, the IBM POWER5™. In addition, we propose the use of hardware resource allocation as a way to balance high performance computing applications. Finally, we propose two new scheduling mechanisms that are able to transparently and successfully balance applications in real systems using the hardware resource allocation. On the soft real-time domain, we propose a hardware extension to the existing explicit resource allocation hardware and, in addition, two software schedulers that use the explicit allocation hardware to improve the schedulability of tasks in a soft real-time system.

In this thesis, we demonstrate that system performance improves by making the software aware of the mechanisms to control the amount of resources given to each running thread. In particular, for the high performance computing domain, we show that it is possible to decrease the execution time of MPI applications biasing the hardware resource assignation between threads. In addition, we show that it is possible to decrease the number of missed deadlines when scheduling tasks in a soft real-time SMT system.

Acknowledgements

I would like to thank my parents, Rita Vieira and Lindomar Wessler Boneti, for their great support though my entire life. Very few people know that we come from a very simple ground and the effort we had to put, as a family, in order to reach the stability in which we live today.

I would also like to thank Alice, for being at my side during this time of hard work. Her understanding and support were key to the conclusion of this thesis.

I must thank the people at IBM T.J. Watson that, together with my advisors, helped and guided me during a very decisive part of this research. In special, but not exclusively, many thanks to the team with whom I worked more closely: Chen-Yong Cher, Alper Buyuktosunoglu, Pradip Bose and Jaime Moreno. Special thanks to Jose Brunheroto and his wife, Gloria.

A person to whom I am deeply grateful is Yale Patt. I believe it is not exaggerated to say that I wouldn't have done this research if it wasn't for him. In addition to keeping me on the track, he was a great professor in more than one class and provided great guidance at moments when I would otherwise be hopeless. It is important to stress the contribution of Patt and Mateo. Their value not only as researchers but as persons is immense.

Of course, this work would never be finished if it wasn't for my great advisors, Francisco J. Cazorla, Roberto Gioiosa and Mateo Valero, who guided me in the dark and taught me a lot about research and working discipline. Their experience and hard work is responsible for this thesis as much as all the years and nights I dedicated to the research.

In addition, I should thank my friends that supported me across almost all continents. They kept feeding me with renewed confidence when I lost all mine. Many persons kept me going and happy in those many years: Dominic, Ana Virgínia, Mateus, Fabio, Bona, Geneviève, Ritvars and so many others should be thanked here. Some of them gave me academic advices, other gave me comforting, other helped me discover Barcelona and its surrounding. With each one of them I can remember different histories and different tales that helped to make these years so unique.

In special Carlos Villavieja and Isaac Gelado helped me in many different occasions since I first arrived in Spain. Being great friends and fellow students, they shared every step of the many phases of life I had since I started in the DAC program. It is by sharing small pieces of our lives that we created strong bounds. I still remember how they

“risked their lives” to help me move my furniture from one apartment to another, or how Carlos kindly let me help him demolish the walls of his new apartment. Isaac fed me with chorizo and orujo del pueblo and kept me company in may days of August or weekends when few other students were in the campus. He was also one of the only crazy enough students to like the idea of staying the whole Saturday working in the UPC until so late that we could go out to the discos directly from the office.

Finally, I would like to thank all the crew from the DAC secretary for their support. In special, Trini, who had to hear from me so many times asking for all kinds of documentation for all kinds of means: visas, grants, thesis deposit, etc. Still, she gave me a great support and made my life much easier during the years I spent in the department.

Contents

List of Figures

List of Tables

Abbreviations

CMP	**C**hip **M**ulti-**P**rocessor
DLRB	**D**ynamic **L**oad and **R**esource **B**alancer
DRA	**D**ual-objective **R**esource **A**llocation
EDF	**E**arliest **D**eadline **F**irst
ERA	**E**xplicit **R**esource **A**llocation
HPC	**H**igh **P**erformance **C**omputing
HPCSched	**HPC Sched**uler
hs	**H**ardware-**S**cheduler
IPC	**I**nstructions **P**er **C**ycle
LLF	**L**east **L**axity **F**irst
LPT	**L**ow **P**riority **T**hread
LVP	**L**ow-**V**ariability **P**erformance
MCT	**M**ost **C**ritical **T**hread
MinRa	**Min**imum **R**esource **A**llocation
MT	**M**ulti-**T**hreaded
NCT	**N**on **C**ritical **T**hread
OS	**O**perating **S**ystem
RA	**R**esource **A**llocation
RC	**R**emaining **C**omputation
SU	**S**erial **U**tilization
TTD	**T**ime **T**o **D**eadline
SMT	**S**imultaneous **M**ultithreading
SRA	**S**ingle-objective **R**esource **A**llocation
WCET	**W**orst **C**ase **E**xecution **T**ime

Dedicated to my parents: Rita Vieira and Lindomar Boneti.

Chapter 1

Introduction

As the process technology evolves and the number of available transistors in a chip increases, limitations in exploitation of the instruction level parallelism and power constraints created a trend where modern processors started to execute multiple simultaneous execution flows. Simply increasing the frequency of a superscalar processor tends to increase the power consumption beyond today's acceptable limits and, therefore, thread-level parallelism has become a common strategy to improve performance.

Multithreaded[1] (MT) processors have widespread use in almost every class of computer system. They offer high performance at a low cost by sharing processor's internal hardware resources among multiple execution flows or threads. In one extreme of the spectrum, in SMT processors, the threads share most of the processor's internal hardware resources, while in the other extreme, in a CMP processor, they typically share cache levels and memory bandwidth. Levels of resource-sharing are often combined, as for instance, the IBM POWER5™ processor is both SMT and CMP: a POWER5 chip has two cores, where each core has two SMT threads.

In addition to the achievement of higher throughput, MT processors have good performance/cost ratio as they often present simpler cores replicated in the chip, which are easier to design than more complex bigger cores. Furthermore, they commonly have better watt per committed instruction ratio than large super-scalar processors.

In this domain several researchers have proposed many hardware improvements to maximize a number of metrics [68][77][76][75][13][19][14][11]. Hardware is often tailored to share the processor's internal resources in order to maximize throughput or fairness.

[1] In this thesis, we refer to multithreaded processors as being any kind of processor that executes multiple threads at the same time. Simultaneous Multithreading (SMT), fine-grain multithreaded, coarse-grain multithreaded or Chip Multi-Processor (CMP) are examples of MT processors.

1.1 Background on resource allocation

One of the major drawbacks of MT processors is that, in most cases, the way the internal hardware resources are split between the different execution flows or threads is not specified by the software, but implicitly decided by the hardware. In other words, such systems have implicit resource allocation. For instance, the sharing in the caches may be decided by the cache replacement policy [22][66], while the fetch bandwidth may be split using a multithreading fetch policy like icount [76], FLUSH [75] or FLUSH++ [13].

To better illustrate the problem, take for instance the fetch policy, which decides how the instructions are fetched from the running threads in a SMT processor. The fetch policy determines implicitly how the internal resources, like the renaming registers or the instruction queue entries, are allocated. A common characteristic of many proposed fetch policies is that they try to increase the processor's throughput and/or fairness [59] by stalling or flushing instructions from threads presenting L2 misses [75][55]. They target to increase the overall system performance by seamlessly controlling the flow of a thread.

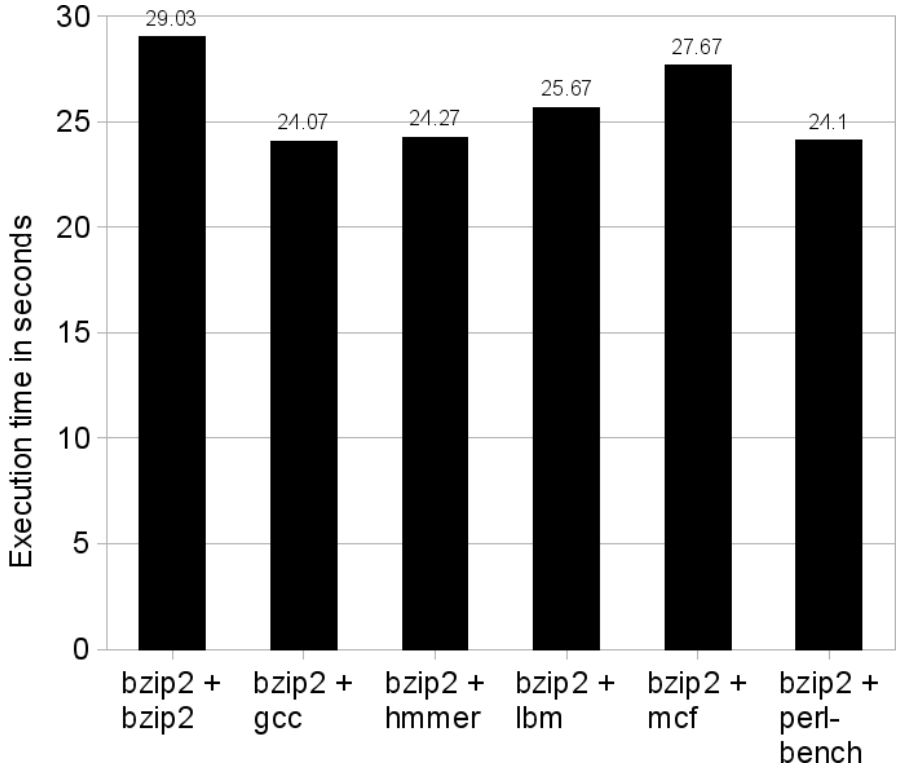
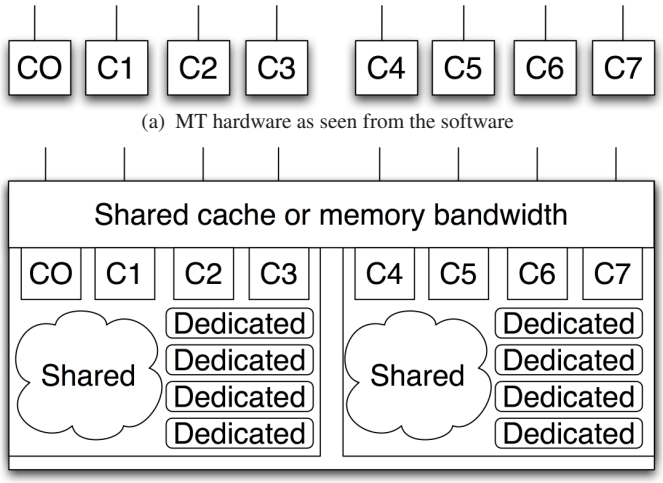


FIGURE 1.1: Execution time of bzip2 when coscheduled with other benchmarks.

MT's internal hardware resource sharing creates undesirable behaviors like unpredictable performance or interference between the concurrent executing threads. Such interference can influence the performance negatively or positively, depending on the nature of the workload and the underlying architecture. As a consequence, the execution time of a program can be very hard to estimate. For instance, Figure 1.1 shows the execution time of bzip2 on a POWER5 processor when coscheduled with other SPEC2006 benchmarks on the same core. As we can see, its execution time variates from 24.07 seconds, when running with gcc, to 29.03 seconds when running with another copy of bzip2.



(a) MT hardware as seen from the software

(b) MT hardware as it is in reality

FIGURE 1.2: Difference between the perception of an MT processor and reality

Figure 1.2 shows how an MT processor is perceived by the software layer and how it may really be. To the software layer, each one of the threads in a multithreaded processor is generally perceived as an independent processing unit and the system considers them as having the same processing power, or the same amount of resources (Figure 1.2(a)). However, the contexts in a MT processor share some common resources, like some cache levels for CMP or internal functional units and fetch bandwidth for SMT (Figure 1.2(b)). The interaction between simultaneous execution flows is not taken into consideration outside of the hardware.

Another problem of this "lack of communication" between the software and the underlying architecture is the following: a program with high *O*perating *S*ystem (OS) priority may receive less hardware resources than a program with low OS priority. For instance, in a processor with *FLUSH* instruction fetch policy, if a program with high OS priority have a large number of cache misses, it will receive less hardware resources than a program with low OS priority and low cache miss rates coscheduled at one of

the processor's hardware contexts. We call this problem a hardware-software priority inversion[2].

To deal with these problems, a new class of MT processors have been proposed in the literature. These proposals allow *E*xplicit *R*esource *A*llocation (ERA) [15], where the OS can specify or bias the hardware resource sharing to meet specific targets.

In academic research, proposals of explicit resource allocation provide "levers" through which the OS can control the processor internal resource allocation [18], or even feedback mechanisms through which the OS can fine tune the resource allocation [17]. Furthermore, the authors argue that it is important to control several layers of the resource sharing in order to provide guarantees of performance to the programs sharing the hardware resources [63].

Typically, the idea behind the explicit resource allocation mechanisms is that the software layer should specify a given amount or percentage of resources to be used by a thread. Then, a possible organization is that for every instruction, as soon as its required resources are known (usually at the decode stage) the hardware updates the list, or counter, of the used resources for the thread that fetched this instruction. If this thread reached its allocation limit, then it stops fetching or, depending on the proposal, takes another action like flushing this thread's instructions from the pipeline. Finally, the used resources list is also updated to reflect the resources that are no longer being used (usually at the commit stage). In that way, the hardware exercise a more direct control over the resource distribution. The measured resources, the control mechanism and the actions taken when threads exceed their quota of resources varies according to the proposals.

There has been extensive research on hardware with explicit resource allocation and, some of these research have been reflected in the industry. As for instance, the first processor to allow the software to bias the internal hardware allocation, the IBM POWER5 processor, presents two levels of thread prioritization: the first level provides dynamic prioritization through hardware, while the second level is a software-controlled priority mechanism that allows a thread to specify a priority value from 0 to 7. Currently, this mechanism is only used in few cases in the software platforms [60] even if it can provide significant improvements on several metrics. In fact, the POWER5 software-controlled hardware priorities are only used to decrease the resources of a context when there is no useful computation being done [60] (Section 2.4.3). We argue that it is mainly due to the fact that there are no previous works aimed at the characterization of the effects of this mechanism.

[2]This problem is not to be confused with the classic scheduling priority inversion problem.

Even if the majority of the commercial processors are in some form MT, currently, most of the operating system task scheduling is done as if the logical processors were fully independent, ignoring the interactions between the concurrent threads in a core, or between cores of a chip. Very few optimizations exist today, such as considering cache locality when re-scheduling a task, trying to keep a process within a core or a thread that shares the same L2 cache where it was scheduled before. In addition, most of the MT processors still have implicit resource allocation. In other words, there is no way to explicitly allocate internal processor's resources to one thread in detriment of another.

Although there are many hardware proposals to control multithreaded (SMT or CMP) processors hardware resource sharing, in our view, there is a lack of integration between the software and the hardware layers that sometimes yields to both parts working in opposite directions. Furthermore, a given hardwired metric may not suit the needs of the software when running a specific problem.

In this thesis we show that sometimes it may be necessary to sacrifice a processor's throughput to decrease the execution time of a program. Such situations not only happen in user desktops, but in many other domains, like real-time systems or in the the High Performance Computing (HPC) domain. For instance, in the case of a real-time system, the ideal behavior may be that drawing the user interface receives just enough resources so it does not slows down important packets being processed.

We believe that allowing the system software to control the resource sharing of the multithreaded hardware will allow the development of several techniques that improve the appropriated targets of several application domains. This thesis narrows the gap between ERA hardware and system software mechanisms by proposing scheduling techniques for ERA processors and an improved explicit resource allocation hardware for soft-real time.

1.2 Thesis Contributions

The main goal of this thesis is to propose software and hardware mechanisms for HPC and soft real-time systems, using software controlled hardware resource-allocation to improve system's performance. By doing so, we fill the gap between the explicit resource allocation hardware previously proposed in the literature and the software-level prioritization.

The main contributions of this thesis are:

- We characterize the first real HPC processor featuring a software-controlled hardware prioritization, the IBM POWER5. Furthermore, we present the infrastructure needed to use this processor's hardware support for explicit resource allocation.

  This work resulted in the following publication:

  * Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Chen-Yong Cher, Alper Buyuktosunoglu and Mateo Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *International Symposium on Computer Architecture (ISCA)*. Beijing, China. June 21-25, 2008.

- We present, for the first time, the idea of resource allocation as a means of balancing high performance computing applications.

  – We propose a dynamic process scheduler for the Linux kernel that automatically and transparently balances HPC applications according to their behavior.

  – We present an application-level load balancer that is easily deployed across a large number of machines and provides automatic and transparent load balancing for HPC applications.

  The overall work on load-balacing resulted in the following publications:

  * Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Julita Corbalan, Jesus Labarta and Mateo Valero. Balancing HPC Applications Through Smart Allocation of Resources in MT Processors. In *International Parallel & Distributed Processing Symposium (IPDPS)*. Miami, Florida, USA. April 14-18, 2008.

  * Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla and Mateo Valero. A Dynamic Scheduler for Balancing HPC Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Austin, USA. November 15-21, 2008.

  * Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla and Mateo Valero. Using resource allocation to balance HPC applications. To appear in: *Parallel and Distributed Computing*, IN-TECH, Viena, Austria ISBN978-3-902613-45-5, 2009. (Book Chapter)

- In the soft real-time domain, we propose *Resource Aware* extensions for two well known schedulers, the *Earliest Deadline First* and the *Least Laxity First*. These extensions are respectively called RA-EDF and RA-LLF. In addition, we propose

a novel hardware support that allows to dynamically improve a secondary metric, in this case throughput, while guaranteeing a minimal resource allocation.

This work resulted in the following publication:

* Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa and Mateo Valero. Soft Real-Time Scheduling on SMT Processors with Explicit Resource Allocation. In *International Conference on Architecture of Computing Systems (ARCS)*. Dresden, Germany. February 25,2008. Lecture Notes in Computer Science. Volume 4934/2008

1.3 Thesis Presentation

This thesis is organized as follows:

- Chapter 1 presents the research field and problem matters, along with the objectives of this research. Furthermore, it presents the contributions of this research and the thesis structure.
- Chapter 2 presents the characterization of the effects of the software-controlled hardware prioritization of POWER5 on several different workloads. It presents two application case studies targeting different performance metrics and shows the circumstances where a background thread can be ran transparently without effecting the performance of the foreground thread.
- Chapter 3 shows that by appropriately using the software-controlled prioritization mechanism, it is possible to reduce the imbalance in parallel applications transparently to the user and reducing the total execution time.
- Chapter 4 presents a dynamic process scheduler for the Linux kernel that automatically and transparently balances HPC applications according to their behavior.
- Chapter 5 presents an application-level load balancer that is easily deployed across a large number of machines.
- Chapter 6 targets the scheduling of soft real-time tasks on an explicit resource allocation processor. It presents *Resource Aware* extensions for two well known schedulers and a novel hardware support that allows to dynamically improve a secondary metric while guaranteeing a minimal resource allocation.
- Chapter 7 presents the conclusions of this thesis.

Chapter 2

Software-Controlled Priority Characterization of POWER5 Processor

2.1 Introduction

In the first part of this thesis, we focus on the High Performance Computing (HPC) domain, in which some processors allow the control of the assignment of hardware resources. Having a commercially available processor with explicit resource allocation allows us to evaluate our proposals on a real hardware, with a real operating system.

We start with a detailed analysis of the IBM POWER5^{TM} processor, which will be used for our experiments in the remaining of the HPC sections.

The IBM POWER5 is a dual-core processor, where each core runs two threads. Threads share many resources such as the Global Completion Table (GCT or reorder buffer), the Branch History Table (BHT) and the Translation Lookaside Buffer (TLB).

It is well known that higher performance is realized when resources are appropriately balanced across threads [49][50][69]. An IBM POWER5 system appropriately balances the usage of resources across threads with mechanisms in hardware [35][49]. Moreover, POWER5 employs a mechanism, through software/hardware co-design, that controls the instruction decode rate with eight priority levels. Its main motivation is to address instances where unbalanced thread priority is desirable. Several examples can be enumerated such as idle thread, thread waiting on a spin-lock, software determined

non-uniform balance and power management [50][69]. Software-controlled priority[1] can significantly improve both throughput and execution time depending on the application type.

In the literature, a wide range of mechanisms has been proposed on dynamically balancing of resources to improve SMT performance. Most of these proposals focus on the instruction fetch policy as the means to obtain such balancing [32][75]. In addition to the instruction fetch policy, other mechanisms explicitly prioritize shared resources among threads to improve throughput, fairness [19][23] and Quality of Service [17]. While these studies do not correspond to the software prioritization mechanism of POWER5, they could justify the use of the mechanism.

Nevertheless, the prioritization mechanism provided by POWER5 is rarely used among the software community and, even in these rare cases, the prioritization mechanism is mainly used for lowering the priority of a thread. For instance, the Linux kernel version 2.6.23 exploits the software-controlled priorities in few cases to reduce the priority of a processor that is not performing any useful computation. Moreover, Linux makes the assumption that the software-controlled priority mechanism is not used by the programmer and resets the priority to the default value at every interrupt or exception handling point.

Currently, the lack of quantitative studies on software-controlled priority limit their use in real world applications. In this chapter, we provide the first quantitative study of the POWER5 prioritization mechanism. We show that the effect of thread prioritization depends on the characteristics of a given thread and the other thread it is coscheduled with. Our results show that, if used properly, software-controlled priorities may increase overall system performance, depending on the metric of interest. Furthermore, this study helps Linux and other software communities to tune the performance of their software by exploiting the software-controlled priority mechanism of the POWER5 processor.

The main contributions of this chapter are:

1. We provide a detailed analysis of the effect of the POWER5 prioritization mechanism on execution time of applications with a set of selected micro-benchmarks that stress specific workload characteristics. We show that:

- Threads executing long-latency operations (i.e., threads with a lot of misses in the caches) are less effected by priorities than threads executing short-latency

[1] POWER5 software-controlled priorities are independent of the operating systems concept of process or task priorities.

operations (i.e. cpu-bound threads). For example, we observe that increasing the priority of a cpu-bound thread could reduce its execution time by 2.5x over the baseline. Increasing the priority of memory-bound threads causes an execution time reduction of 1.7x when they are run with other memory-bound threads.

- By reducing the priority of a cpu-bound thread, its performance can decrease up to 42x when running with a memory-bound thread and up to 20x when running with another cpu-bound thread. In general, improving the performance of one thread involves a higher performance loss on the other thread, sometimes by an order of magnitude. However, decreasing the priority of a long-latency thread has less effect on its execution time compared to a cpu-bound thread. For example, decreasing the priority of a memory-bound thread increases its execution time by 22x when running with another memory-bound thread, while increases less than 2.5x when running with the other benchmarks. In Section 2.5.3 we show how to exploit this to improve the overall performance.

- For the micro-benchmarks used in this chapter, the IPC throughput of the POWER5 improves up to 2x by using software-controlled priorities.

- We also show that a thread can run transparently, with almost no impact on the performance of a higher-priority thread. In general, foreground threads with lower IPC are less sensitive to a transparent thread.

2. We present two application case studies that show how priorities in POWER5 can be used to improve two different metrics: aggregated IPC and execution time.

- In the case of a batch application where the main metric is throughput, the performance improves up to 23.7%.

- In the case of an unbalanced MPI parallel application, execution time reduces up to 9.3% by using priorities to re-balance its resources.

To our knowledge, this is the first quantitative study showing how software-controlled prioritization of POWER5 effects performance on a real system. Since other processors like the IBM POWER6^{TM} [53] present a similar prioritization mechanism, this study can be significantly useful for the software community.

This chapter is organized as follows: Section 2.2 presents the related work. Section 2.3 describes the POWER5 resource balancing in hardware and the software-controlled priority mechanisms. Section 2.4 presents our evaluation environment, and Section 2.5 shows our results and their analysis. Finally, Section 2.6 concludes this work.

2.2 Related Work

In the literature a wide range of mechanisms have been proposed to prioritize the execution of a thread in a SMT processor. Many of these proposals focus on the instruction-fetch policy to improve performance and fairness in SMT processors, while other focus on explicitly assigning processor resources to threads.

Instruction Fetch Policies: An instruction fetch (I-fetch) policy decides how instructions are fetched from the threads, thereby implicitly determining the way processor resources, like rename registers or issue queue entries, are allocated to the threads. Many existing fetch policies attempt to maximize throughput and fairness by reducing the priority, stalling, or flushing threads that experience long latency memory operations [19][75]. Some other fetch policies focus on reducing the effects of misspeculation by stalling on hard-to-predict branches [51][58].

Explicit Resource Allocation: Some of the mechanisms explicitly allocate shared processor resources targeting throughput improvements [19][23]. Other resource allocation mechanisms provide better QoS guarantees for the execution time by ensuring a minimum performance for the time critical threads [20][17].

2.3 The POWER5 Processor

IBM POWER5 [43] processor is a dual-core chip where each core runs two threads [50]. POWER5 employs two levels of control among threads, through resource balancing in hardware (Section 2.3.1), as well as software-controlled prioritization (Section 2.3.2).

2.3.1 Dynamic hardware resource balancing

POWER5 provides a dynamic resource-balancing mechanism that monitors processor resources to determine whether one thread is potentially blocking the other thread execution. Under that condition, the progress of the offending thread is throttled back, allowing the sibling thread to progress. POWER5 considers that there is an unbalanced use of resources when a thread reaches a threshold of L2 cache or TLB misses, or when a thread uses too many GCT (reorder buffer) entries.

POWER5 employs one of the following mechanisms to re-balance resources among threads: 1) It stops instruction decoding of the offending thread until the congestion

clears (Stall). 2) It flushes all of the instructions of the offending thread that are waiting for dispatch and stopping the thread from decoding additional instructions until the congestion clears (Flush). Moreover, the hardware may temporarily adjust the decode rate of a thread to throttle its execution.

2.3.2 Software-controlled priorities

The number of decode cycles assigned to each thread depends on the software-controlled priority. The enforcement of these software-controlled priorities is carried by hardware in the decode stage. In general, the higher the priority, the higher the number of decode cycles assigned to the thread.

Let us assume that two threads, a primary thread (*PThread*) and a secondary thread (*SThread*), are running on one of the two cores of the POWER5 with priorities PrioP and PrioS, respectively. Based on the priorities, the decode slots are allocated using the following formula:

$$R = 2^{|PrioP-PrioS|+1} \tag{2.1}$$

Table 2.1 shows the possible values of R and how many decode slots are assigned to the two threads as the difference between ThreadA's and ThreadB's priority moves from 0 to 4. Notice that R is computed using the difference of priorities of *PThread* and *SThread*, *PrioP-PrioS*. At any given moment, the thread with higher priority receives R-1 decode slots, while the lower priority thread receives the remaining slot. For instance, assuming that *PThread* has priority 6 and *SThread* has priority 2, R would be 32, so the core decodes 31 times from *PThread* and once from *SThread* (more details on the hardware implementation are provided in [35]). The performance of the process running as *PThread* increases to the detriment of the one running as *SThread*. On the case where both threads have the same priority, $R = 2$, and therefore, each thread receives one slot, alternately.

TABLE 2.1: Decode cycles allocation in the IBM POWER5 with different priorities

Priority difference (PrioP-PrioS)	**R**	**Decode cycles** for A	**Decode cycles** for B
0	2	1	1
1	4	3	1
2	8	7	1
3	16	15	1
4	32	31	1

In POWER5, the software-controlled priorities range from 0 to 7, where 0 means the thread is switched off and 7 means the thread is running in Single Thread (ST) mode (i.e., the other thread is off). The supervisor or operating system can set six of the eight priorities ranging from 1 to 6, while user software can only set priority 2, 3 and 4. The Hypervisor can always use the whole range of priorities.

As described in [35] and [43], the priorities can be set by issuing an `or` instruction in the form of `or X,X,X`, where `X` is a specific register number. This operation only changes the thread priority and performs no other operation. If it is not supported (when running on previous POWER processors) or not permitted due to insufficient privileges, the instruction is simply treated as a `nop`. Table 2.2 shows the priorities, the privilege level required to set each priority and how to change priority using this interface.

TABLE 2.2: Software-controlled thread priorities in the IBM POWER5 processor.

Priority	**Priority level**	**Privilege level**	**or-nop instruction**
0	Thread shut off	Hypervisor	-
1	Very low	Supervisor	`or 31,31,31`
2	Low	User/Supervisor	`or 1,1,1`
3	Medium-Low	User/Supervisor	`or 6,6,6`
4	Medium	User/Supervisor	`or 2,2,2`
5	Medium-high	Supervisor	`or 5,5,5`
6	High	Supervisor	`or 3,3,3`
7	Very high	Hypervisor	`or 7,7,7`

The behavior of the software-controlled thread prioritization mechanism is different when one of the threads has priorities 0 or 1 as shown in Table 2.3 [35][43]. For instance, when both threads have priority one, instead of being considered as difference 0 and perform as having no prioritization, the processor runs in low-power mode, decoding only one instruction every 32 cycles.

TABLE 2.3: Resource allocation when the priority of any of the threads is 0 or 1

PThread	***SThread***	**Action**
> 1	> 1	Decode cycles are given to the two threads as explained above
1	> 1	*SThread* gets all the execution resources; *PThread* takes what is left over
1	1	Power save mode; both *PThread* and *SThread* receive 1 of 32 decode cycles
0	> 1	Processor in ST mode. *SThread* receives all the resources.
0	1	1 of 32 cycles are given to *SThread*
0	0	Processor is stopped

2.4 Evaluation Methodology

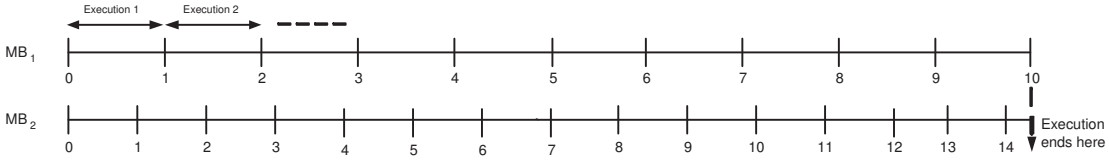


FIGURE 2.1: Example of application of the FAME methodology. In this example Micro-Benchmark 1 takes longer than Micro-Benchmark 2.

In order to explore the capabilities of the software-controlled priority mechanism in the POWER5 processor, we performed a detailed set of experiments. Our approach consists of analyzing the processor as a black-box, observing how the performance of a workload changes as we increase or reduce the priority of threads.

In a SMT processor the performance of one process not only depends on the processor architecture but also on the other processes running at the same time on the same core and their specific program phases. Under such conditions, evaluating all the possible programs and all their phase combinations is simply not feasible. Moreover, when it comes to a real system evaluation, with the several layers of the running software, the OS interferences and all the asynchronous I/O services, the problem becomes even worse.

For this reason, we use a set of micro-benchmarks that stresses a particular processor characteristic. While this scenario is not typical with real applications, this is one of the best ways to understand the mechanism under evaluation. It provides a uniform characterization based on specific program characteristics that can be mapped into real applications. With real applications it would be impossible to attribute fine-grain performance gain/loss to the prioritization mechanism due to applications own variability.

2.4.1 Running the experiments

This chapter uses the FAME (FAirly MEasuring Multithreaded Architectures) methodology [78][79]. In [78] the authors state that the average accumulated IPC of a program is representative if it is similar to the IPC of that program when the workload reaches a steady state. The problem is that, as shown in [78][79], the workload has to run for a long time to reach this steady state. FAME determines how many times each benchmark in a multi-threaded workload has to be executed so that the difference between

the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value.

For the experimental setup and micro-benchmarks used in this chapter, in order to accomplish a MAIV of 1%, each micro-benchmark must be repeated at least 10 times. In our experiments we run two workloads, hence each experiment ends when both threads re-execute at least 10 times. Note that, at this point the fastest thread might already execute more than 10 times. Figure 2.1 shows an example where the second benchmark is faster than the first. In this example, while the MB_1 ($MicroBenchmark_1$) executes 10 times, MB_2 executes 14 times. It is important to note that the execution time difference is not constant. For instance, if we change the software-controlled priorities, MB_2 may execute faster or slower, and therefore we must guarantee that both threads execute a minimum number of repetitions. In our experiments, the average execution time for a thread is estimated as the total accounted execution time divided by the number of *complete* repetitions. For example, in Figure 2.1, the total execution time of MB_2 is measured until it completes the 14th iteration and the time for the remaining incomplete iteration is discarded.

Furthermore, as previously shown [34][36], normal software environment can insert significant noise into performance measurements. To minimize such noise, both single-thread and multithreaded experiments were performed on the second core of the POWER5. All user-land processes and interrupt requests (IRQ) were isolated on the first one, leaving the second core as free as possible from noise.

2.4.2 Micro-benchmark

In a multi-threaded architecture, the performance of one process tightly depends on the other process that it is running with. Moreover, the effect of the software-controlled priorities depends on the characteristics of the benchmarks under study. In order to build a basic knowledge of these effects, we developed a set of 15 synthetic micro-benchmarks, each of them stressing a specific processor characteristic. This methodology allows us to isolate independent behaviors of the platform. Furthermore, micro-benchmarks provide higher flexibility due to their simplicity.

We classify the micro-benchmarks in four groups: Integer, Floating Point, Memory and Branch, as shown in Table 2.4. In each group, there are benchmarks with different instruction latency. For example, in the Integer group there are short (`cpu_int`)

and long (`lng_chain_int`) latency operation benchmarks. In the Memory group, `ldint_l2` is a benchmark with all loads always hitting in second level of data cache, while `ldint_mem` has all loads hitting in memory and, hence, missing in all cache levels. As expected, `ldint_l2` has higher IPC than `ldint_mem`. In the Branch group, there are two micro-benchmarks with high (`br_hit`) and low (`br_miss`) hit prediction rate.

All the micro-benchmarks have the same structure. They iterate several times on their loop body and the loop body is what differentiates them. One execution of the loop body is called a micro-iteration. Table 2.4 shows the details of the loop body structures for the micro-benchmarks. The size of the loop body and the number of micro-iterations is specific for each benchmark. The benchmarks are compiled with the `xlc` compiler with `-O2` option and their object code are verified in order to guarantee that the benchmarks retain the desired characteristics.

TABLE 2.4: Loop body of the different micro-benchmarks.

Name	**Loop Body**
`cpu_int`	`a += (iter * (iter - 1)) -` x_i `* iter` : $x_i \in \{1, 2, ..., 54\}$
`cpu_int_add`	`a += (iter + (iterp)) -` x_i `+ iter` : $x_i \in \{1, 2, ..., 54\}$; `iterp = iter -1 + a`
`cpu_int_mul`	`a = (iter * iter) *` x_i `* iter` : $x_i \in \{1, 2, ..., 54\}$;
`lng_chain_int`	`a += (iter * (iter - 1)) -` x_0 `* iter` : $x_i \in \{1, 2, ..., 20\}$ `b += (iter * (iter - 1)) -` x_1 `* iter + a` ... `a += (iter + (iter - 1)) -` x_{10} `* j` ... The cycle of instructions is repeated multiple times, for a total of 50 lines in the loop body.
`br_hit br_miss`	`if (a[s]=0) a=a+1; else a=a-1;` $s \in \{1, 2, ..., 28\}$ a is filled with all 0's for br_hit and randomly (modulo 2) for br_miss

`ldint_l1` `ldint_l2`
`ldint_l3` `ldint_mem`
`ldfp_l1` `ldfp_l2`
`ldfp_l3` `ldfp_lmem`
	`a[i+s] = a[i+s]+1;` where s is set that we always hit in the desired cache level.
	
	In the case of fp benchmarks, a is an array of floats.

`cpu_fp`	`a += (tmp * (tmp - 1.0)) -` x_i `* tmp` —$x_i \in \{1.0, 2.0, ..., 54.0\}$. `(float tmp = iter * 1.0)`

After the first complete set of experiments, where we ran all the possible combinations, we realized that some of the benchmarks behave equally and do not add any further insight to the analysis. Therefore, we present only the benchmarks that provide differentiation for this characterization work. For example, `br_hit`, `br_miss`, `cpu_int_add`, `cpu_int_mul` and `cpu_int` behave in a very similar way. Analogously, the load-integers and load-floating-points do not significantly differ. Therefore, we present only the results for `cpu_int`, `lng_chain_int`, `ldint_l1`, `ldint_l2`, `ldint_mem` and `cpu_fp`.

2.4.3 The Linux kernel

Some of the priority levels are not available in user mode (Section 2.3.2). In fact, only three levels out of eight can be used by user mode applications, the others are only available to the OS or the Hypervisor. Modern Linux kernels (2.6.23) running on IBM POWER5 processors exploit software-controlled priorities in few cases such as reducing the priority of a process when it is not performing useful computation. Basically, it makes use of the thread priorities in three cases:

- The processor is spinning for a lock in kernel mode. In this case the priority of the spinning process is reduced.
- The kernel is waiting for operations to complete. For example, when the kernel requests a specific CPU to perform an operation by means of a `smp_call_function()` and it can not proceed until the operation completes. Under this condition, the priority of the thread is reduced.
- The kernel is running the idle process because there is no other process ready to run. In this case the kernel reduces the priority of the idle thread and eventually puts the core in Single Thread (ST) mode.

In all these cases the kernel reduces the priority of a processor's context and restores it to MEDIUM (4) as soon as there is some job to perform. Furthermore, since the kernel does not keep track of the current priority, and to ensure responsiveness, it also resets the thread priority to MEDIUM every time it enters a kernel service routine (for instance, an interrupt, an exception handler, or a system call). This is a conservative choice induced by the fact that it is not clear how and when to prioritize a processor context and what the effect of that prioritization is.

In order to explore the entire priority range, we developed a non-intrusive kernel patch which provides an interface to the user to set all the possible priorities available in kernel mode:

- We make priority 1 to 6 available to the user. As mentioned in Section 2.3.2, only three of the priorities (4, 3, 2) are directly available to the user. Without our kernel patch, any attempt to use other priorities result in a `nop` operation. Priority 0 and 7 (context off and single thread mode, respectively) are also available to the user through a Hypervisor call.

- We remove the use of software-controlled priorities inside the Linux kernel, otherwise the experiments would be effected by unpredictable priority changes.
- Finally, we provide an interface through the `/sys` pseudo file system which allows user applications to change their priority.

For the experiments described in this chapter, the patch was applied to the standard Linux kernel version 2.6.19.2.

2.5 Analysis of the Results

TABLE 2.5: IPC of micro-benchmarks in ST mode and in SMT with priorities (4,4). *pt* stands for *PThread* and *tt* for total IPC.

Micro benchmark	IPC ST	IPC in SMT (4,4)											
		ldint_l1		ldint_l2		ldint_mem		cpu_int		cpu_fp		lng_chain_int	
		pt	tt	pt	tt	pt	tt	pt	tt	pt	tt	pt	tt
ldint_l1	2.29	1.15	2.31	0.60	0.87	0.79	0.81	0.73	1.57	0.77	1.18	0.42	0.91
ldint_l2	0.27	0.27	0.87	0.11	0.22	0.17	0.19	0.27	0.87	0.25	0.65	0.27	0.72
ldint_mem	0.02	0.02	0.81	0.02	0.19	0.01	0.02	0.02	0.90	0.02	0.39	0.02	0.48
cpu_int	1.14	0.84	1.57	0.59	0.87	0.88	0.90	0.61	1.22	0.65	1.06	0.43	0.86
cpu_fp	0.41	0.41	1.18	0.39	0.65	0.37	0.39	0.40	1.06	0.36	0.72	0.37	0.85
lng_chain_int	0.51	0.49	0.91	0.45	0.73	0.47	0.48	0.43	0.86	0.48	0.85	0.42	0.85

In this section, we show to what extent the prioritization mechanism of POWER5 effects the execution of a given thread and the trade-off between prioritization and throughput.

The following sections use the same terminology as Section 2.3.2. We call the first thread in the pair the "Primary Thread", or *PThread*, and the second thread the "Secondary Thread" or *SThread*. The term PrioP refers to priority of the primary thread, while PrioS refers to the priority of the secondary thread. The priority difference (often expressed as $PrioP - PrioS$) can be positive in which case the *PThread* has higher priority than the *SThread* or negative where the *SThread* has higher priority. The results are normalized to the default case with priorities (4,4).

Table 2.5 presents the IPC values in single thread mode as well as in SMT mode with priorities (4,4). For each row, the column *pt* shows the IPC of the primary thread and *tt* shows the total IPC. For example, the second row presents the case where `ldint_l2` is the primary thread. The IPC ST column shows its single thread IPC value (0.27). The third column present its IPC when running with `ldint_l1` (0.27) and the fourth column shows the combined IPC of `ldint_l1` and `ldint_l2` when running together (0.87).

In the Sections 2.5.1 and 2.5.2 we discuss about the effects of negative and positive prioritization. This effect is not symmetric as it follows the formula 2.1. For instance, at priority +4 a thread receive 31 of each 32 decode slots, which represents an increase of 93.75% of the resources when compared to the baseline, where a thread receives half of the resources. However, at priority -4, a thread receives only one out of 32 decode slots, which represents 16 times less resources.

On the Figures 2.2 and 2.3, the results represent the relative performance of the primary thread shown in the graph's caption when coscheduled with each one of the other benchmarks in the legend. The results are a factor of the baseline case with no prioritization.

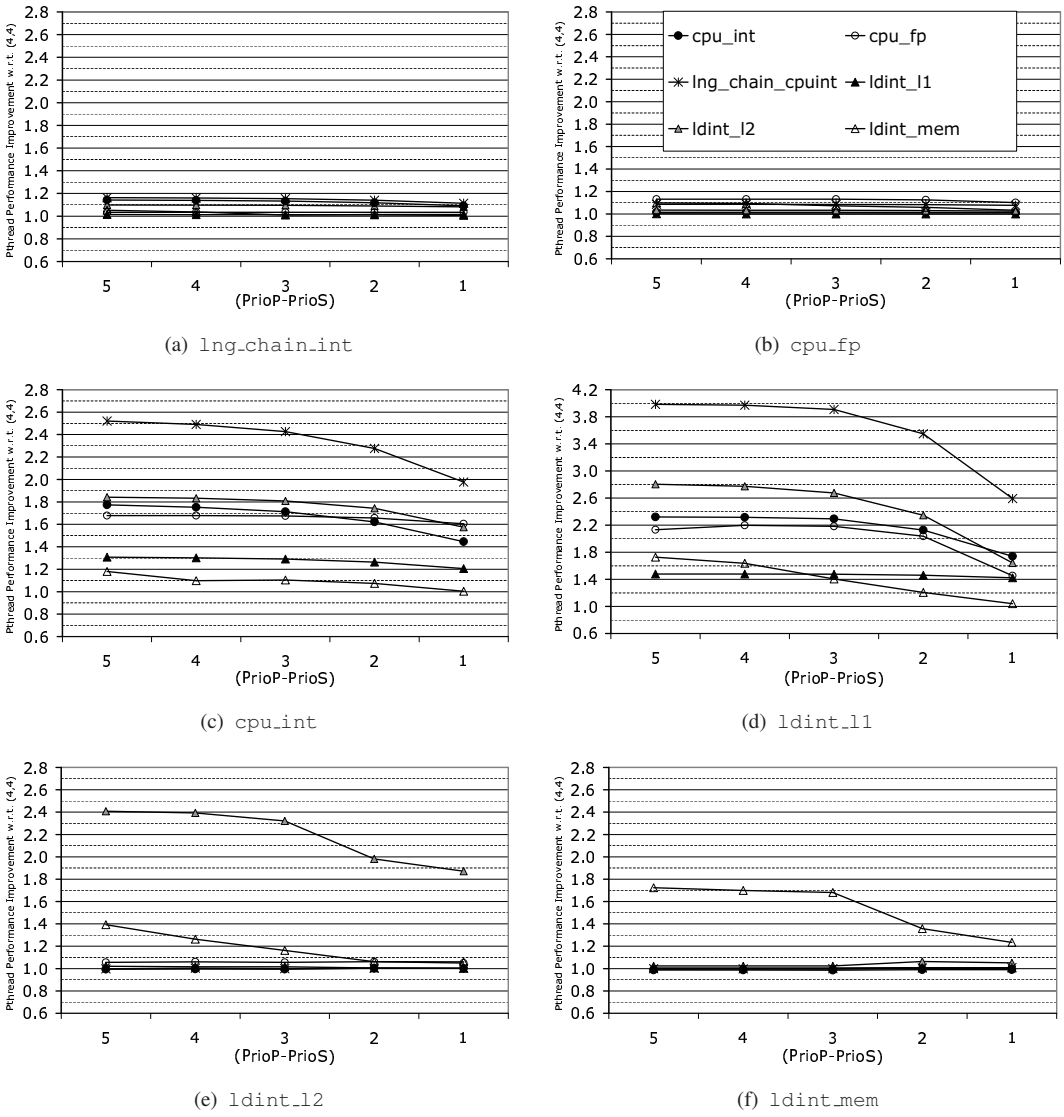
2.5.1 Effect of Positive Priorities

In this section, we analyze the performance improvement of the *PThread* with different *SThreads* using positive priorities (*PrioP* $>$ *PrioS*). Figure 2.2 shows the performance improvement of the *PThread* as its priority increases with respect to the *SThread*. For example, Figure 2.2(c) shows the results when we run `cpu_int` as *PThread*.

In general, the threads that have low IPC and are not memory-bound, such as `lng_chain_int` and `cpu_fp`, benefit less from having high priorities. Memory-bound benchmarks, such as `ldint_l2` and `ldint_mem`, benefit from the prioritization mechanism when they run with another memory-bound thread. This improves performance up to 240% for `ldint_l2` and 70% for `ldint_mem`. On the other hand, high IPC threads, like `cpu_int` and `ldint_l1` are very sensitive to the prioritization as they can benefit from the additional hardware resources. Therefore, their prioritization usually improves the total throughput and increases their performance.

The results show that the memory-bound benchmarks are also effected by the POWER5 prioritization mechanism, when competing with other benchmarks of similar requirements. They are less sensitive than the purely cpu-bound benchmarks, and they only benefit from an increased priority when co-scheduled with other memory-bound benchmarks. As a rule of thumb, memory-bound benchmarks should not be prioritized except when running with other memory-bound benchmark. Section 2.5.3 shows that prioritizing these workloads is often in detriment of the overall system performance.

In addition, a priority difference of +2 usually represents a point of relative saturation, where most of the benchmarks reach at least 95% of their maximum performance. The memory-bound benchmarks represent an exception to this rule, where the largest performance increase occurs from a priority difference of +2 to +3.



(a) `lng_chain_int` (b) `cpu_fp`

(c) `cpu_int` (d) `ldint_l1`

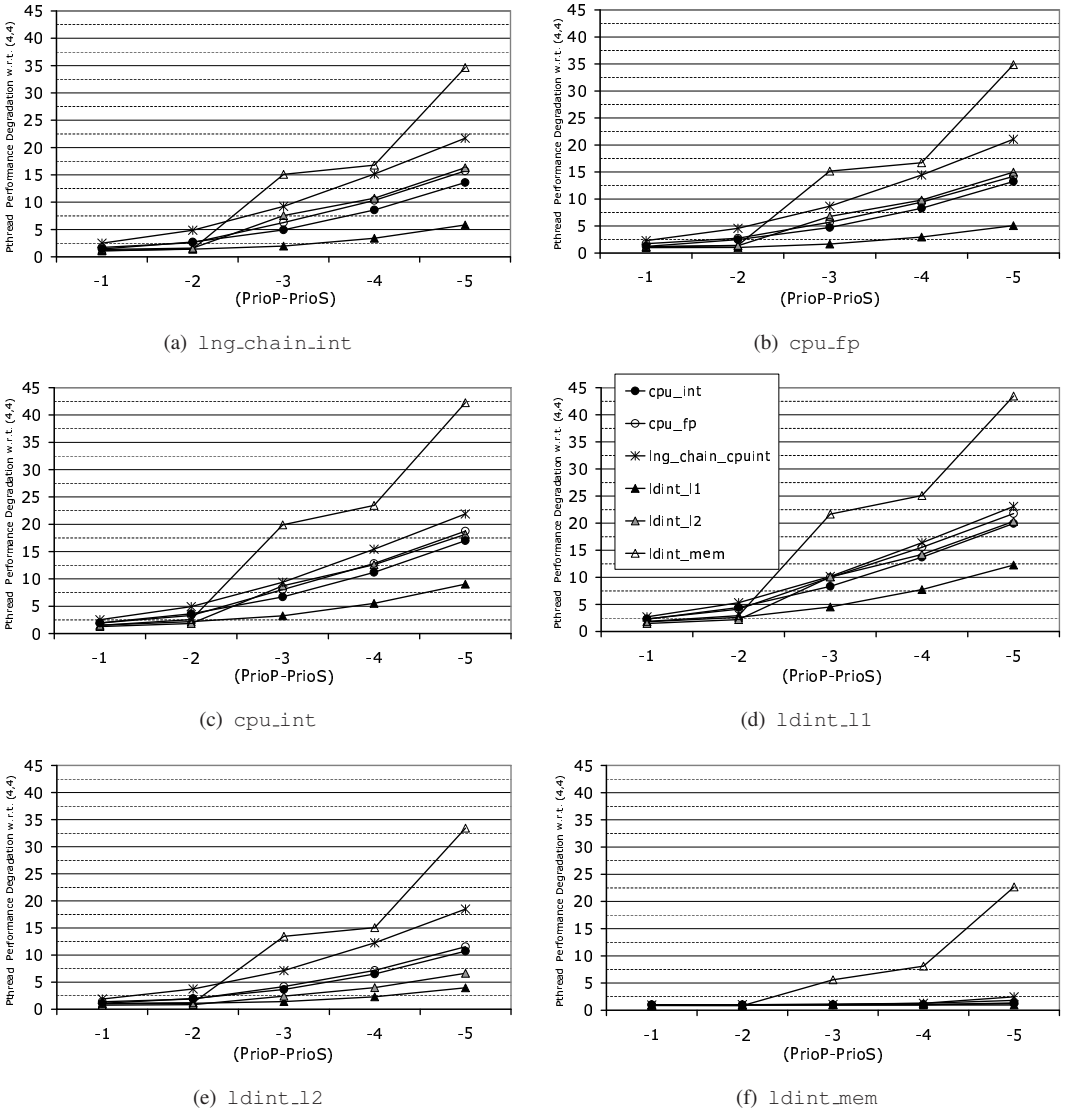
(e) `ldint_l2` (f) `ldint_mem`

FIGURE 2.2: Performance improvement of the *PThread* as its priority increases with respect to the *SThread*. Note the different scale for `ldint_l1`.

2.5.2 Effect of Negative Priorities

In this section, we present the effects of the negative priorities (*PrioP* $<$ *PrioS*) on the micro-benchmarks. Figures 2.3 (a) to (e) show that setting negative priorities heavily impacts the performance of all micro-benchmarks except for `ldint_mem`. The effect of the negative priorities on the performance is much higher than the effect of the positive priorities. While using positive priorities could improve performance up to four times, negative priorities can degrade performance by more than forty times.

Figure 2.3 (f) presents that `ldint_mem` is insensitive to low priorities in all cases other than running with another thread of `ldint_mem`. In general, a thread presenting high



(a) `lng_chain_int` (b) `cpu_fp`

(c) `cpu_int` (d) `ldint_l1`

(e) `ldint_l2` (f) `ldint_mem`

FIGURE 2.3: Performance degradation of the *PThread* as its priority decreases with respect to the *SThread*.

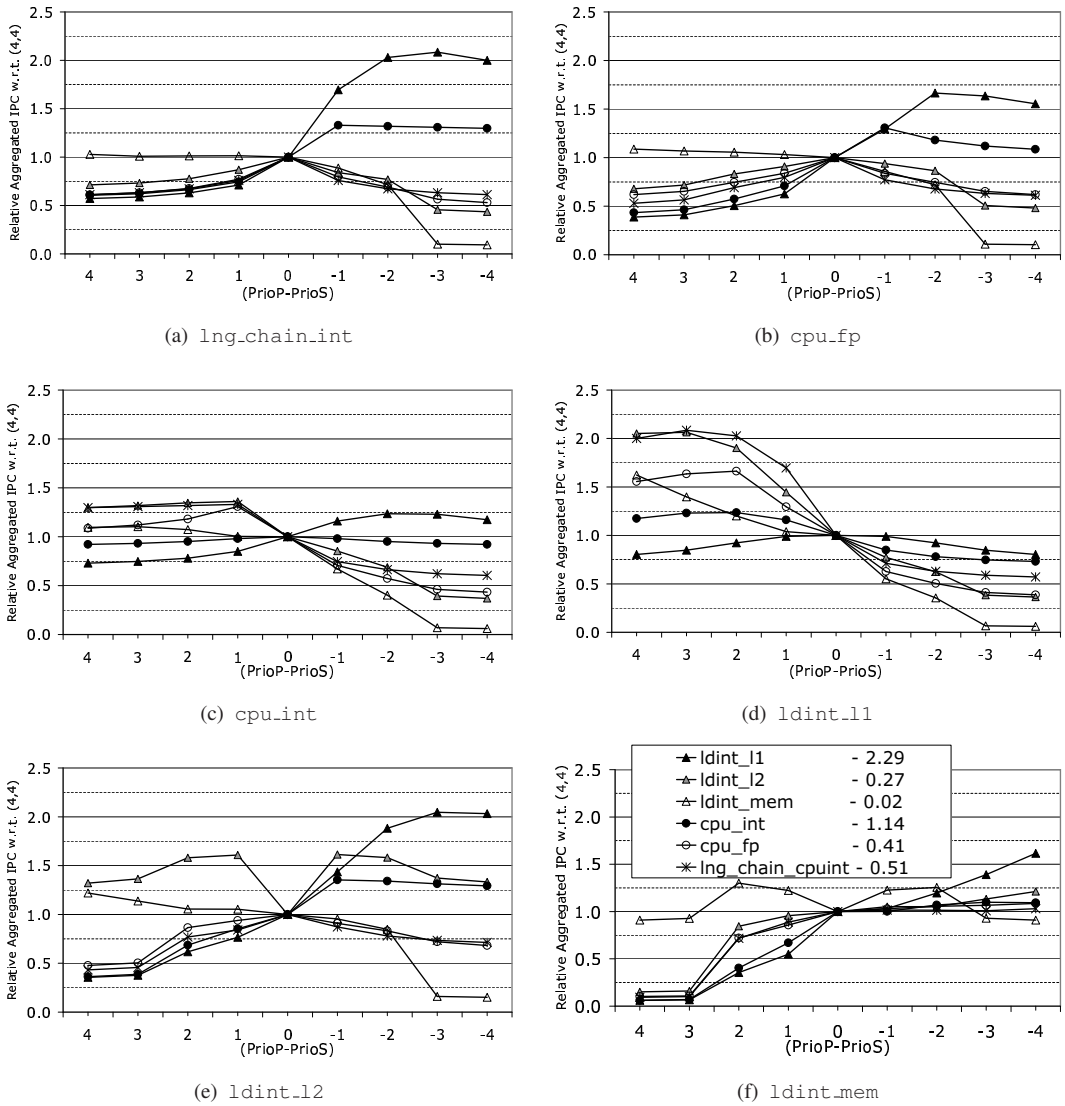
latency memory operation, long dependency chains or slow complex operations is less effected by a priority reduction.

Memory-bound benchmarks are the ones that impact the other threads the most. They also present clear steps of performance impact when the priority difference changes from -2 to -3, and from -4 to -5. The priority difference of -5 is extreme since the *PThread* obtains only the left-overs from the memory thread. In general, a priority difference like -5 should only be used for a transparent background thread in which the performance is not important.

While priority difference of +2 usually yields close to the maximum performance, priority -3 results in a clear delta on performance loss. For memory-bound threads, there

is no significant performance variation from 0 to -2. Considering that priority difference +2, most of the high IPC threads reach 95% of their maximum performance, this suggests that priority differences larger than +/-2 should normally be avoided. Section 2.5.3 shows the additional throughput that can be obtained based on this conclusion.

2.5.3 Optimizing IPC Throughput



(a) `lng_chain_int` (b) `cpu_fp`

(c) `cpu_int` (d) `ldint_l1`

(e) `ldint_l2` (f) `ldint_mem`

FIGURE 2.4: Throughput w.r.t. execution (4,4). The legend shows the single-thread IPC of benchmarks.

POWER5 employs several hardware mechanisms to improve the global throughput, like stalling the decode of the low IPC tasks or flushing the dispatch of threads that

would otherwise decrease the overall performance of the system. The POWER5 built-in resource balancing mechanism is effective in most cases where changing the thread's priorities negatively impact the total throughput.

Even though the baseline is effective, Figure 2.4 shows several cases where the total throughput can be improved up to two times or more. This comes at the expense of severe slowdown of the low priority thread, especially when the low priority thread has low IPC such as `lng_chain_int`. These cases can be exploited for systems where total throughput is the main goal and where the low IPC thread can actually afford the slowdown.

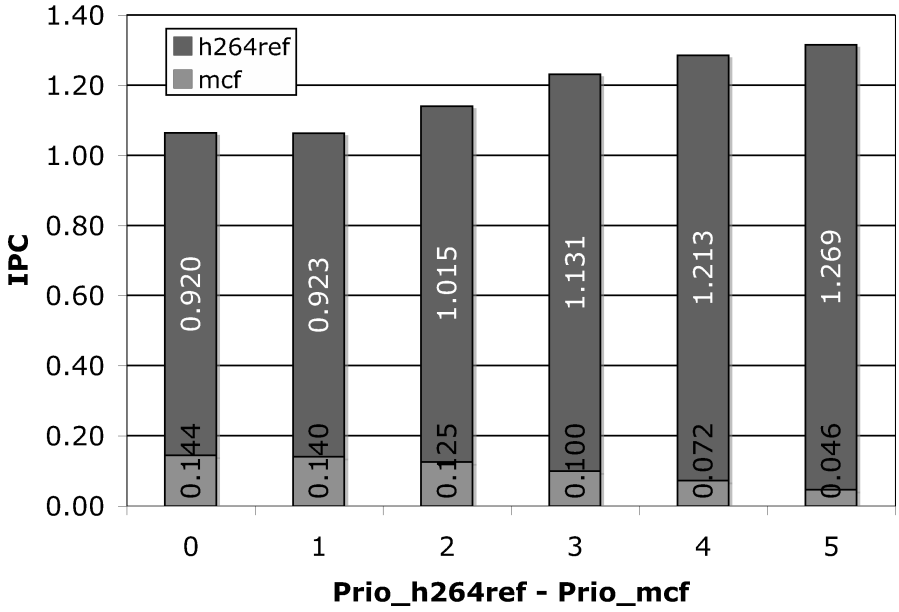
Furthermore, while the performance for the cpu-bound benchmarks increase with their priority, the performance of a memory benchmark remains relatively constant. Using the prioritization mechanism for this combination yields, almost always, significant throughput improvement. In general, we obtain an IPC throughput improvement when we increase the priority of the higher IPC thread in the pair.

2.5.3.1 Case Study

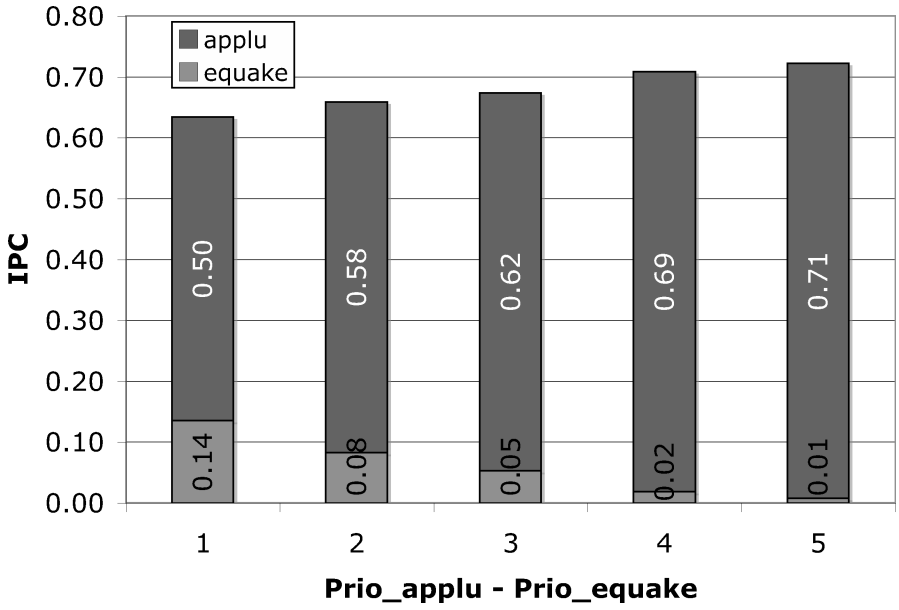
In order to verify whether our findings can be applied to real workloads, this section shows how software-controlled prioritization can improve total IPC. We analyze the behavior of two pairs of SPEC CPU 2000 and 2006 benchmarks [41]. The first one is composed of *464.h264ref* (from now on referred as *h264ref*) and *429.mcf* (from now on referred as *mcf*). The second pair is composed of *173.applu* (from now on referred as *applu*) and *183.equake* (from now on referred as *equake*). We take as the baseline the scenario where they run side by side, on different contexts of the same core, without any type of software-controlled prioritization (i.e., with the same priority). The experiments follow the FAME methodology.

When running with the default priorities (4,4), *h264ref* has an IPC of 0.920 and takes about 3254 seconds to complete, and *mcf* takes 1848 seconds and reaches an IPC of 0.144. The total IPC for this configuration is 1.064. Figure 2.5 (a) shows the performance of both benchmarks as we increase the priority of *h264ref*. We can see that, until priority difference +2, the performance of the *mcf* is reduced by 13.2%, while *h264ref* gains 10.4%. While the gain and the loss are very similar in performance, the overall throughput increases by 7.2%. Further increase in the throughput is possible by degrading low IPC benchmark. The peak IPC is reached when *mcf* runs 32% slower and *h264ref* runs 38% faster than the base case with default priorities. In this case, the overall system performance increases by 23.7%.

For the second pair, with the default priorities, *applu* has an IPC of 0.500 and completes in 240 seconds. *equake* takes 74 seconds and has an IPC of 0.140. Together, they reach a total IPC of 0.630 (Figure 2.5 (b)). In this case, the peak combined IPC is obtained when *applu* receives priority +5. It represents a 14% of improvement when compared to the default case.



(a) h264ref and mcf



(b) applu and equake

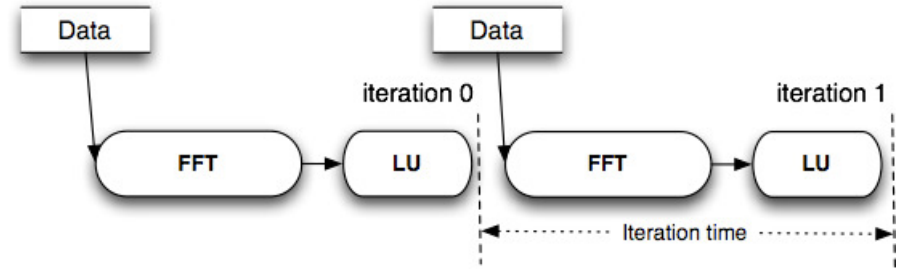
FIGURE 2.5: Total IPCs with increasing priorities

2.5.4 Optimizing execution time

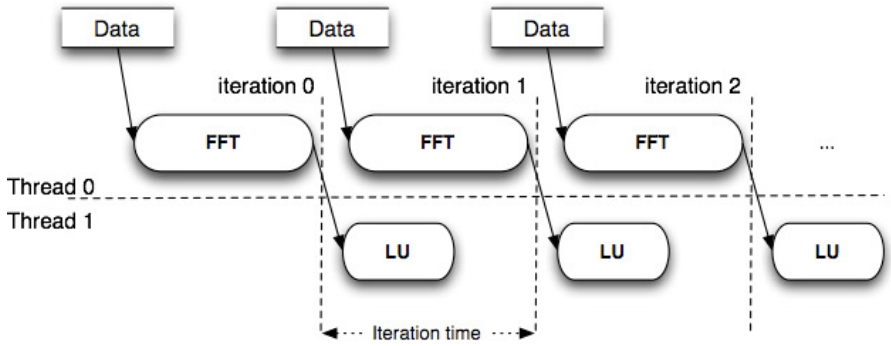
The highest throughput does not always directly translate into the shortest execution time of a whole application [12]. Most of the parallel applications have synchronization points where all the tasks must complete some amount of work in order to continue. Load balancing in parallel application is a hard problem since it is rarely the case where the synchronized tasks finish perfectly at the same time. In other words, usually a task has to wait for other tasks to complete. This could clearly delay the progress of the whole program.

2.5.4.1 Case Study

In this section we present an example where we are able to improve the overall application execution time by using the prioritization mechanism. In this example, we apply a `LU` matrix decomposition over a set of results produced by a Fast Fourier Transformation (`FFT`) for a given spectral analysis problem (Figure 2.6(a)). One possible organization of the problem would create a software pipeline where one thread runs the Fast Fourier Transformation, producing the results that will be consumed by the second thread on the next iteration, by applying LU over parts of this output (Figure 2.6(b)).



(a) Single-threaded execution



(b) Multithreaded execution

FIGURE 2.6: Single-threaded and multithreaded (with a software pipeline) organizations of LU and FFT combination.

In our measurement, the FFT takes for 1.86 seconds in single-thread mode, and the LU takes 0.26 seconds to process its part of the problem. In single-thread mode, the processor would first execute the FFT and then the LU, thus, each iteration would require 2.12 seconds to complete. In the multi-threaded scenario, there is only FFT running in the first iteration to produce the first input of the LU. On the remaining iterations, both threads would be running in parallel and the execution time of an iteration would be the execution time of the longest thread. As we can see on the Table 2.6, when run together in SMT mode, the FFT takes 2.05 seconds and the LU decomposition takes 0.42 seconds. The LU thread would waste 1.63 seconds waiting for the other task to complete. Using the prioritization mechanism, we could increase the priority of FFT so it executes faster, reducing the unbalance.

TABLE 2.6: Execution time, in seconds, of FFT and LU.

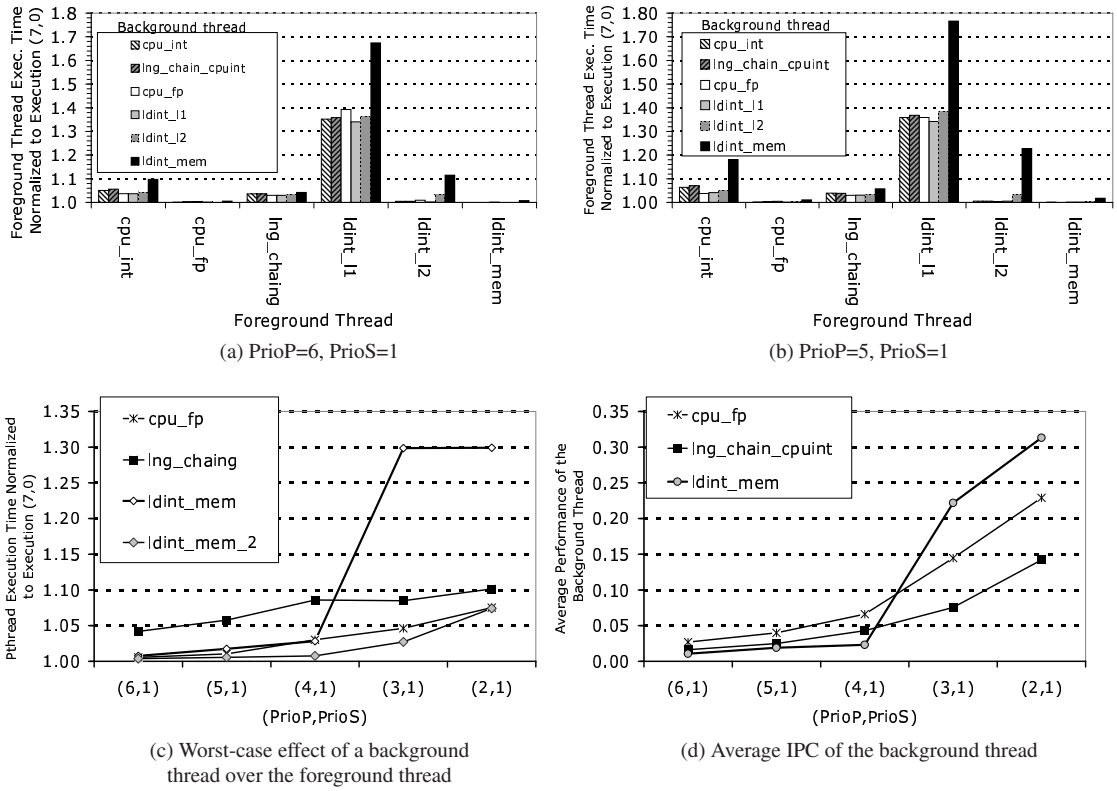
Priority	Priority Difference	FFT exec. time	LU exec. time	Iteration exec. time
single-thread mode	-	1.86 –	– 0.26	2.12
4,4	0	2.05	0.42	2.05
5,4	+1	2.02	0.48	2.02
6,4	+2	1.91	0.64	1.91
6,3	+3	1.87	2.33	2.33

Table 2.6 shows that the best case consists of running with a priority pair (6,4), which yields an iteration execution time of 1.91 seconds. Effectively this represents a 10% improvement when compared to the execution time in single thread mode (where it would be necessary to run the FFT followed by LU) and 9.3% of improvement over the default priorities. On the other hand, by applying too much prioritization, it's possible to inverse the unbalance, which normally represents a performance loss (priority (6,3)).

The idea of using POWER5 prioritization mechanisms to balance real HPC applications, reducing their execution time, will be further explored in the next chapters of this thesis.

2.5.5 Transparent execution

Dorai and Yeung [28] propose transparent threads, which is a mechanism that allows background threads to use resources that a foreground thread does not require for running at almost full speed. In POWER5 this is implemented by setting the priority of the "background" thread to 1 [35].



(a) PrioP=6, PrioS=1

(b) PrioP=5, PrioS=1

(c) Worst-case effect of a background thread over the foreground thread

(d) Average IPC of the background thread

FIGURE 2.7: Primary thread Execution Time with respect to Single-Thread when *SThread* has priority 1

Figure 2.7 shows the effect of background threads over foreground threads when a foreground thread runs with priority 6 (Figure 2.7 (a)) and with priority 5 (Figure 2.7 (b)). We observe that the most effected threads are `ldint_l1`, `cpu_int` and `ldint_l2`, when they are running with a memory-bound background thread.

Figure 2.7(c) presents the maximum effect that a background thread causes on the other foreground threads (`ldint_l2`, `cpu_fp`, and `lng_chain_int`) as we reduce its priority from 6 to 2. In the figure, the different foreground threads run with `ldint_mem` in background as it proved to be the worst case for all combinations.

For `cpu_fp` and `lng_chain_int` the effect of the background thread increases linearly as we reduce the priority from 6 to 2 until about 10% of their ST performance. This is not the case for `ldint_mem` that suffers a sudden increment when its priority is 3 or 2. In the chart, the label 'ld_int_mem_2' represents the performance of the `ldint_mem` when it runs as a foreground thread and the `ldint_mem` is not the background thread. The graph shows that the effect that any other micro-benchmark causes on `ldint_mem` is about 7%. We can conclude that, unless running with another copy

of itself, `ldint_mem` can always run as foreground thread without much performance degradation.

Finally, Figure 2.7(d) shows the performance of the background threads. Each point represents the average for all background threads: for example, the point `ldint_mem` (6,1) represents the average performance of the background thread in the experiments (`ldint_mem`, `cpu_int`), (`ldint_mem`, `cpu_fp`), (`ldint_mem`, `lng_chain_int`), (`ldint_mem`, `ldint_l1`), and (`ldint_mem`, `ldint_mem`) using priorities (6,1). We can observe that in the worst performance degradation case (under 10%) for `cpu_fp`, the background threads obtain an IPC of 0.23. For the `lng_chain_int` benchmark this IPC is 0.15.

In general, we can establish that the high-latency threads are the best candidates for foreground thread and the worst background thread. They suffer little impact from a background thread, but heavily effect the performance when running in background. Furthermore, threads with very high performance easily get effected by other threads (see `ldint_l1` on Figure 2.7 (a)). They may not be suitable to run with a background thread.

2.6 Conclusions

The IBM POWER5 processor presents two levels of thread prioritization: the first level provides dynamic prioritization through hardware, while the second level is a software-controlled priority mechanism that allows a thread to specify a priority value from 0 to 7. Currently, this mechanism is only used in few cases in the software platforms even if it can provide significant improvements on several metrics. We argue that it is mainly due to the fact that there are no previous works aimed at the characterization of the effects of this mechanism.

In this chapter we perform an in-depth evaluation of the effects of the software-controlled prioritization mechanism over a set of synthetic micro-benchmarks, specially designed to stress specific processor characteristics. We present the following conclusions from our micro-benchmarks. First, workloads presenting a large amount of long-latency operations are less influenced by priorities then the ones executing low-latency operations (i.e., integer arithmetic). Second, it is possible, by using the prioritization mechanism, to improve the overall throughput up to two times, in very special cases. However, those extreme improvements often imply drastic reduction of the low IPC thread's performance. On less extreme cases, it is possible to improve the throughput by 40%.

And third, we show that, instead of using the full spectrum of priorities, only priorities up to +/-2 should be used, while "extreme" priorities should be used only when the performance of one of the two threads is not important.

In addition, we present three case studies where priorities can be used to improve the total throughput by 23.7%, the total execution time by 9.3% or to have a background thread. Finally, we conclude that the prioritization mechanism in POWER5 is a powerful tool that can be used to improve different metrics. This work opens a path into broader utilization of a software/hardware co-design that allows better balancing of the underlying hardware resources among the threads.

In the following chapters, we explore the use of software controlled hardware resource allocation as a way to balance high performance computing applications. Chapter 3 presents a proof of concept of this technique with a deeper study than the one presented in Section 2.5.4.1.

Chapter 3

Balancing HPC Applications Through Smart Allocation of Resources in MT Processors

3.1 Introduction

High Performance Computing (HPC) applications are usually *Single Process-Multiple Data* (SPMD) and are implemented using an MPI or an OpenMP library. In MPI applications, all the processes execute the same code on different data sets and use synchronization primitives (such as barriers or collective operations) to coordinate their work. Since the processes execute the same code, they are supposed to reach their synchronization points roughly at the same time. However, this is not always the case.

Some applications among those running on MareNostrum, the supercomputer installed at the Barcelona Supercomputing Center (BSC), suffer from *imbalance*, i.e. the execution time of the processes in the parallel application is not the same (in Section 3.2 we will see some causes of applications' imbalance). Therefore, if a process runs for longer than the others belonging to the same application, all the other processes have to wait for that process to complete its execution. During this time the CPUs of the waiting processes are idle, thus, not performing any useful job. As an example, let us assume that one process has to complete its execution while all the other processes are waiting for it to reach the synchronization point; then, in MareNostrum, up to 10239 processor may be idle, resulting in a significant loss of performance and waste of resources.

Resource sharing makes multi-threaded processors have good performance/cost and performance/power consumption ratios [5], two desirable characteristics for a Supercomputer. As a consequence, most of the current Supercomputers already use processors with some multi-threaded features [4].

Usually, software has no control over how processor resources are distributed among running threads in multi-threaded processors. For example, in an SMT processor the *instruction fetch policy*, decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor resources are allocated to the threads. This is an undesirable characteristic that makes the execution time of programs unpredictable [17]. In order to alleviate this problem, recently, some processor vendors have equipped their MT processors with mechanisms that allow the software to control processor's internals resource allocation, and thus, control application's speed. Our view is that these mechanisms open new opportunities to improve applications performance as they offer fine-grain ways to control the progress of applications by allocating or deallocating processor resources to them.

This chapter is a first step towards that direction. We show how re-assigning hardware resources in a multi-threaded processor can reduce the imbalance in parallel applications, and hence improving performance. In particular we propose a way to regain balance assigning more hardware resources to processes that computes for more time, reducing their execution time and, thus, the waiting time of all the other processes belonging to the same HPC application. This solution is transparent to the users: since the solution is at Operating System (OS)/hardware level, users do not need to know the processor's implementation details at compile time nor to adapt their programming model in order to use our proposed solution. To the best of our knowledge, this is the first time that such a solution is implemented in a real machine.

We explored this idea experimentally on a real system with a MT processor, the IBM POWER5™ [69]. The POWER5 is a dual-core, 2-way SMT processor that allows us to change the way hardware resources are assigned to the core's contexts by means of a *thread context priority* (or hardware thread priority[1]) that controls the number of resources each context receives. This machine runs a Linux kernel that we had to modify in order to allow the HPC application to exploit the advantage of re-assigning the processor's resources. We performed several experiments with MPI applications:

1. We started from a micro-benchmark (Metbench), developed at BSC, where we introduce some artificial imbalance.

[1]The hardware thread priorities mentioned here are independent of the operating systems concept of thread priority.

2. In the second experiment, we ran the widely used the BT-MZ NAS [64] benchmark; this version suffers of imbalance.

3. Finally, we used a real application running on MareNostrum, SIESTA [2].

Our results show an improvement of 18% for the NAS benchmark and 8.1% for SIESTA. In addition, our results also show that this mechanism of controlling hardware resources is a powerful tool that, if used incorrectly, may lead to significant performance loss. Moreover, non-HPC applications may benefit differently from re-assigning hardware resources or not at all.

The rest of this chapter is organized as follows: Section 3.2 shows the imbalance problem in HPC applications; Section 3.3 presents similar works in the same area; Section 3.4 introduces our solution based on smart allocation of hardware resources; Section 3.5 shows our set of experiments on the IBM POWER5 system for our micro-benchmark, a standard benchmark and a real application; finally Section 3.6 provides our conclusion and future work.

3.2 Imbalance in HPC applications

HPC applications are usually SPMD, which means that every process executes the same code on different data. For example, let us assume that an HPC application is performing a matrix-vector multiplication and that each process receives a sub-matrix and the part of the vector required to compute the sub-matrix by vector multiplication. If the matrix can be divided into homogeneous parts (i.e., they require the same amount of time to be processed), all the processes in the parallel application would finish, ideally, at the same time.

However, the data set could be very different: let us say that, in the previous example, the matrix is sparse or triagonal, hence, the time required to process the data sub-set could vary as well. In this scenario the amount of time required to complete the sub-matrix by vector multiplication depends on the number of non-zero values present in the sub-matrix. In the extreme case, one process could receive a full sub-matrix while another an empty one. It is clear that the former process requires much more time to reach the synchronization point than the latter.

We classify the sources of imbalance in two main classes: *intrinsic* and *extrinsic* factors of imbalance.

3.2.1 Intrinsic imbalance

We refer to *intrinsic imbalance* as the imbalance an application experiences because of data (for example a sparse matrix) or algorithm (master-slave architecture) imbalance. The causes for the imbalance are internal to the application's code, input set or both.

The intrinsic imbalance could be caused by several factors, here we point some of them out:

Input set: As we already said, this scenario happens when a process has a small input set to work on while another has a large amount of data to process.

Domain: Iterative methods approximate the solution of a problem (for example, Partial Differential Equations, PDE) with a function in some domain starting from an initial condition. The domain is divided in several sub-domains and each process computes its part of the solution. At the end of every iteration, the error made in the approximation is computed and, eventually, another iteration is to be started. If the function in some part of the domain is smooth, only few iterations are required to converge to a good approximation. Conversely, if the function has several picks in the sub-domain, more iterations are necessary to find a good solution and/or more points in the domain have to be considered during the computing phase.

Data exchanging: During their execution, processes may require to exchange data among themselves. If the two peers are on the same node, the latency of the communication is small; if a process needs to exchange data with a neighbor on another node the latency is large, even larger if the destination process is far away in the network.

In all the previous cases,the application might result to be imbalanced.

3.2.2 Extrinsic imbalance

Even if both the application's algorithm and the input set are balanced, the execution of the parallel application could still be imbalanced. This is caused by external factors that slow some processes down (but not others). For example, the Operating System (OS) might decide to run another process (say a kernel daemon) in place of the MPI process running on one CPU. Since that MPI process is not able to run all the time while the others are running, it takes more time to complete, making all the other processes wait for it.

Those external factors are the sources of *extrinsic imbalance*. There may be several causes for the imbalance:

OS noise: The CPU is used by the OS to perform services such as handling interrupts, reclaiming memory, assigning memory on demand, etc. The OS noise has been recognized as one of the major source of extrinsic imbalance [36, 65, 74]. A classical example is the interrupt annoyance problem present in Intel processors: all the interrupts coming from external devices are routed to CPU0, therefore, the OS noise caused by executing the interrupt handlers on CPU0 is higher than the noise on the other CPUs.

User daemons: It is common that HPC systems also run profile or statistic collectors together with the HPC application. These activities could steal computing power from one process, delaying it.

Network topology: Exchanging data between processes in the same sub-network is faster than exchanging data between processes in different sub-network; the same rule applies to processes communicating inside a NUMA domain versus processes running in different NUMA domains. In general, if the job scheduler has placed processes that need to communicate "far away", their communication latency could increase so much that the whole application will be affected.

An expert programmer could reduce the intrinsic imbalance in the application. However, this is not an easy task, as the imbalance can be caused by the algorithm, but it can also by the input data set, changing distribution and intensity according to different inputs. Even worse is the case of extrinsic imbalance: those factors are neither under the control of the application nor of the programmer and there is no straightforward way to solve this problem. Thus, it is clear that a mechanism that aims to solve the imbalance of an application should be transparent to the user, regardless of the programming model, libraries or input set.

3.3 Related work

Traditional solutions to attack the problem of load imbalance in HPC applications typically use dynamic data re-distribution. For OpenMP applications load balancing may be performed using some of the existing loop scheduling algorithms that assigns iterations to threads dynamically [7]. MPI applications are much more complex because data communications are defined explicitly in the algorithm by programmers. Static approaches for distributing data using sophisticated tools have been proposed: for example, METIS [1] analyzes data and tries to find the best data distribution. These

approaches achieve good performance results but have the drawback that they must be repeated for each input data set and architecture. Dynamic approaches have also been proposed in the literature (Schloegel et al. [67] and Walshaw et al. [80]). The authors try to solve the load-balancing problem of irregular applications by proposing mesh repartitioning algorithms and evaluating the convenience of repartitioning the mesh or adjusting it.

Processing re-distribution is another approach that consists of assigning more resources to those processes that compute for more time. In the case of OpenMP, this can be useful when using nested parallelism, assigning more software threads to those groups with high load [29]. The case of MPI is much more complex because the number of processes is statically determined when starting the job (in case of malleable jobs), or when compiling the application (in case of rigid jobs). This problem has been also approached through hybrid programming models, combining MPI and OpenMP. Huang and Tafti [42] balance irregular applications by modifying the computational power rather than using the typical mesh redistribution. In their work, the application detects the overloading of some of its processes and tries to solve the problem by creating new software threads at run time. They observe that one of the difficulties of this method is that they do not control the operating system decisions which could oppose their own ones.

Concerning the use of SMT architectures for HPC applications, several studies [21, 24] show that Hyper-Threading (the SMT implementation of Intel Processors) improve performance for some workloads. However, for other workloads there are many conflicts when accessing shared resources, creating a negative impact on the performance. In [24] the study is performed for MPI applications while in [21] the study focuses in OpenMP applications. In [21] the authors propose a mechanism that, given a multiprocessor machine with Hyper-Threading processors, dynamically deactivates the Hyper-Threading in some processors in order to improve the performance of the workload under study.

Our proposal is orthogonal to both the thread re-distribution and the dynamically activating Hyper-Threading. Let us assume that we want to run an HPC application on a cluster having several IBM POWER5 processors. The proposal in [21] can be used to determine in which cores SMT has to be deactivated. For those cores with the SMT feature active, our proposal can be used to select the appropriate hardware priority to reduce imbalance. Compared with thread-distribution, our contribution can be seen as low level solution for load balancing.

3.4 Our Proposal

Balancing a HPC application by hand is a time-consuming task and may require quite a lot of effort. In fact, the programmer has to distribute the data among the processes considering the way the algorithm has been implemented and the correctness of the application. Moreover, this work has to be done for each application and, likely, every time the input changes. As we will see later, our proposal is transparent to the user and independent from the applications or the input set.

With the arrival of MT architectures, and in particular those that allow the software to control processor's resource allocation, new opportunities arise to mitigate the problem of imbalance in HPC systems. This is mainly due to the fact that the software is allowed to exercise a fine control over the progress of tasks, by allocating or deallocating processor resources to them. Such a transparent, fine-grain control cannot be achieved by previous solutions for load imbalance; in fact, even if a lot of processors with shared resources have been introduced in the market since early 90s, very few of them allow the software to control how internal resources are shared. We think that allowing the software to control how to assign shared resources is a key factor for HPC systems. In this view, having MT processor able to provide such mechanism will be essential for improving the performance of HPC systems.

Our solution for balancing HPC applications consists of assigning more hardware resources to the most compute-intensive processes (the bottleneck). Giving this process more hardware resource shall decrease its execution time and, since this process is the bottleneck of the application, the execution time of the whole MPI application.

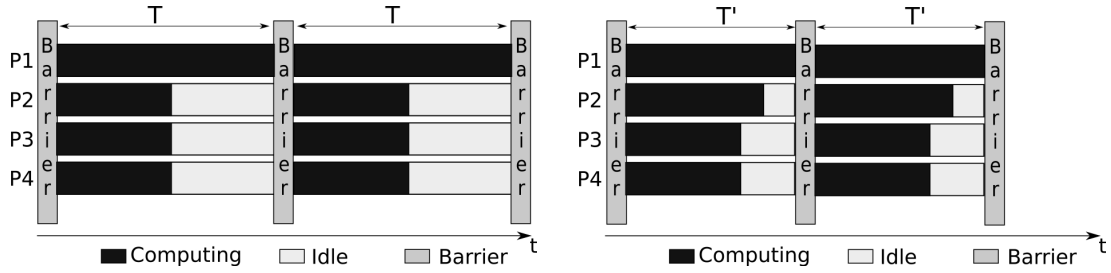
Clearly the underlying processor has to support this capability to re-assign processor resources among running threads. Currently, multi-threaded processors like the IBM POWER5 [69] and POWER6 [53] or the Cell processor [44, 45] provide such a capability with their thread priority mechanisms: the higher the priority of one context, the higher the amount of resources it receives. In this chapter, we focus on the IBM POWER5 but our idea is general and can be also applied to other MT processors that allow the OS to the allocation of processor's resources (for example, partitioning a shared L2 cache in a multi-core CPU [62, 66]). The IBM POWER5 processor is used, among others, by ASC Purple, installed at the Lawrence Livermore National Laboratory[2].

[2]The 3rd supercomputer in the Top500 list of 06/2006, the 11th at the list of 11/2007.

More details about the POWER5 prioritization mechanism are available in Chapter 2. Specifically, the details about the prioritization interface to the software are available in Section 2.3.2.

In addition, recall that, in the previous chapter (Section 2.4.3), we showed that not all the priorities are available from the user-level and why a special kernel patch was needed to enable the use of the full spectrum of POWER5 software-controlled hardware priorities. For the proposal in the current chapter, we employ the same patch developed to perform the characterization in Chapter 2 and described in Section 2.4.3.

In this moment the patch only provides a mechanism to set all the priorities (available at OS level) from user applications. It is responsibility of the user applications (or run time systems) to balance the system load using this interface. This is the first step to prove that our proposal is a good solution for the problem of imbalance in HPC. Our next step, explored in the following chapters of this thesis, will be to have systems that dynamically change the priority of the running processes so that more resources are assigned to the most intensive processes automatically.



(a) Imbalanced HPC application

(b) More balanced HPC application

FIGURE 3.1: Expected effect of the proposed solution ($T' < T$).

We should point out that increasing the performance of one process by giving it more hardware resources, does not come for free. In fact, at the same time, the performance of the other process running on the same core, therefore sharing the resources with the former process, reduces. Figure 3.1 shows a synthetic example that illustrates this case: in Figure 3.1(a) $P1$ shares resources with $P2$, while $P3$ shares them with $P4$; $P2$, $P3$ and $P4$ take the same amount of time to reach their synchronization point but $P1$ takes much more time. As a result $P2$, $P3$ and $P4$ are idle for a long time. In Figure 3.1(b) $P1$ uses more hardware resources and its execution time decreases; $P2$'s execution time, instead, increases since it runs with less hardware resources. Still $P2$ has enough "spare time" and its slowdown does not affect the application's performance because it is not the bottleneck. On the other hand, the performance improvement of $P1$ directly translates into a performance improvement for the whole application, as it is possible to see confronting Figures 3.1(a) and 3.1(b).

Finally we would like to point out that we made no assumption on what kind of application, the programming model or the input set the programmer has to use. Our only assumption regards the underlying processor, which must provide a shared resource control mechanism. Besides that, our solution is at OS level and completely transparent to the users, who are free to use the MPI, OpenMP or whatever programming model or library they wish. Moreover the approach is dynamic and the amount of resources assigned to a process can change according to the input set provided to the application.

3.5 Experimental Results

In order to validate our proposal we performed experiments on an IBM OpenPower 710 server, which has one POWER5 processor.

Since MPI is the most common protocol, we tested our proposal using MPI applications (in the experiments we used the MPI-CH 1.0.4p1 implementation of MPI protocol).

We present our results for three different cases: Section 3.5.1 shows how the IBM POWER5 priority mechanism works using our micro-benchmark (Metbench); Section 3.5.2 provides details on how we used the hardware priorities to balance a widely used benchmark (BT-MZ) and improve its performance. Finally Section 3.5.3 presents the results for a real application frequently executed on MareNostrum (SIESTA).

In order to present experiments in a simple way, we used as metric the total execution time of the application. We used PARAVER [52], a visualization and performance analysis tool developed at CEPBA, to collect data and statistics and to show the trace of each process during the tests.

3.5.1 Metbench

Metbench (Minimum Execution Time Benchmark) is a suite of MPI micro-benchmarks developed at BSC whose structure is representative of the real applications running on MareNostrum. Metbench consists of a *framework* and several *loads*. The framework is composed by a *master* process and several *workers*: each worker executes its assigned load and then waits for all the others to complete their task. The role of the master is to maintain a strict synchronization between the workers: once all the workers have finished their tasks, the master eventually starts another iteration (the number of iterations to perform is a run time parameter). The master and the workers only exchange data

during the initialization phase and use an `mpi_barrier()` to get synchronized. In the traces shown in this section, the master thread corresponds to the first thread and is not balanced as it will be always idle, waiting for the conclusion of all worker threads.

One of the goals of Metbench is to allow researchers at BSC to understand the performance and capabilities of a processor or a cluster. In order to do that, we developed several loads, each one stressing a different processor resource (the Floating Point Unit, the L2 cache, the branch predictor, etc) for a given amount of time. Most of these loads are based on the micro-benchmarks presented in Section 2.4.2.

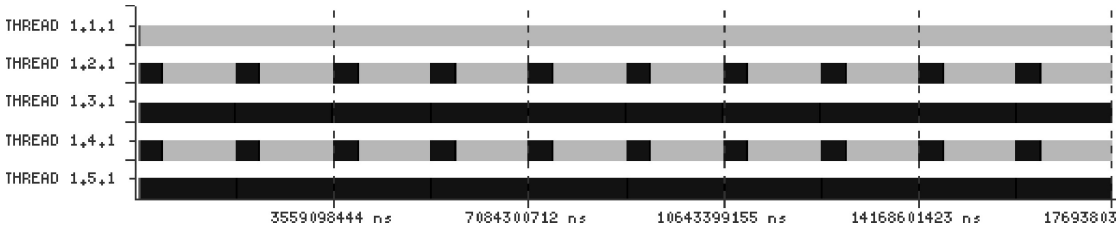
In this experiment we introduce imbalance in the MPI application by assigning to a worker a larger load than the one assigned to the worker on the same core. In this way, the faster worker will spend most of its time waiting for the slower worker to process its load. As we will see in Section 3.5.2 and Section 3.5.3 this scenario is quite common for both standard benchmarks and real applications. Figure 3.2 shows the effect of the proposed solution on Metbench. Each horizontal line represents the activity of a process and each color represents a different state: dark bars show computing time while grey bars show waiting time (at the end of each computation phase there is a black bar that represents statistical operations). In this example, processes $P1$ (the master), $P2$, and $P3$ are mapped to the first core of the POWER5, while processes $P4$ and $P5$ are mapped to the other core. The x-axis represents time.

Case A: Figure 3.2(a) represents our reference case, i.e., the MPI application is running with default priorities (4). As we can see from figure 3.2(a) Metbench shows a great imbalance: more specifically, processes $P1$ and $P3$ spend most of their time waiting for processes $P2$ and $P4$ to complete their computing phase.

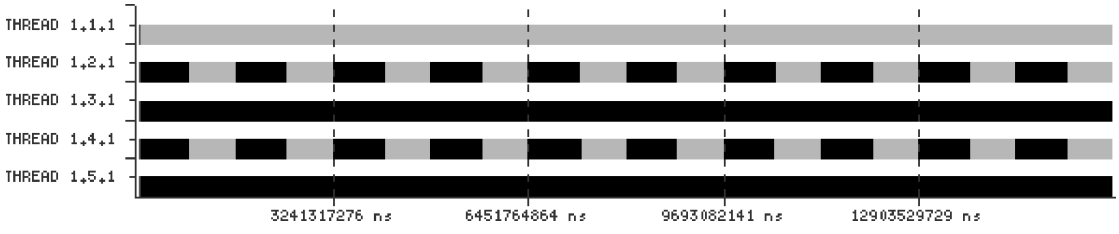
Case B: Using our solution we increased the priority of $P2$ and $P4$ (the most computing intensive processes) up to 6, while the priority of $P1$ and $P3$ are set to 5 (remember from Section 2.3.2 that what really matters is the difference between the thread priorities, here $P1$ and $P3$ are running with less priority than in Case A). Figure 3.2(b) shows how the imbalance has been reduced, also reducing the total execution time (from 81.64 sec to 76.98 sec, 5.71% of improvement).

Case C: Then we tried to reduce again the amount of hardware resources assigned to $P1$ and $P3$, hoping to speed $P2$ and $P4$ up. Indeed, we obtained an even more balanced situation where all the processes compute for (roughly) the same amount of time. The total execution time reduces to 74.90 sec (8.26% of improvement over Case A).

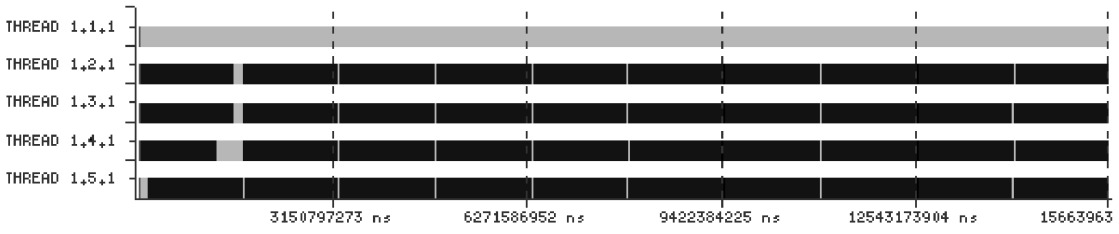
Case D: Next, we reduced again the amount of resources given to $P1$ and $P3$. As we can see from Figure 3.2(d) we reversed the imbalance, i.e., now $P2$ and $P4$ are so



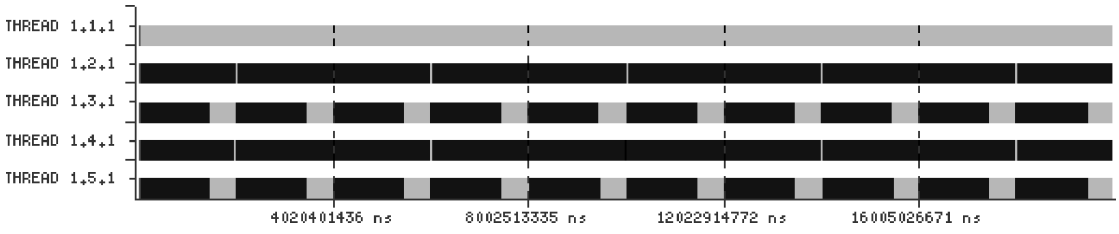
(a) Metbench Case A



(b) Metbench Case B



(c) Metbench Case C



(d) Metbench Case D

FIGURE 3.2: Effect of the proposed solution on Metbench. Each trace represents only some iterations of the application.

much faster than $P1$ and $P3$ that they spend most of their time waiting. As a result the execution time (95.71 sec) increases.

Case D shows an interesting property of the IBM POWER5 hardware priority mechanism: the hardware thread priority implementation is a powerful tool but the performance of the penalized process can be reduced much more than linearly (in fact, exponentially), thus, it could become the new bottleneck.

TABLE 3.1: Metbench balanced and imbalanced characterization

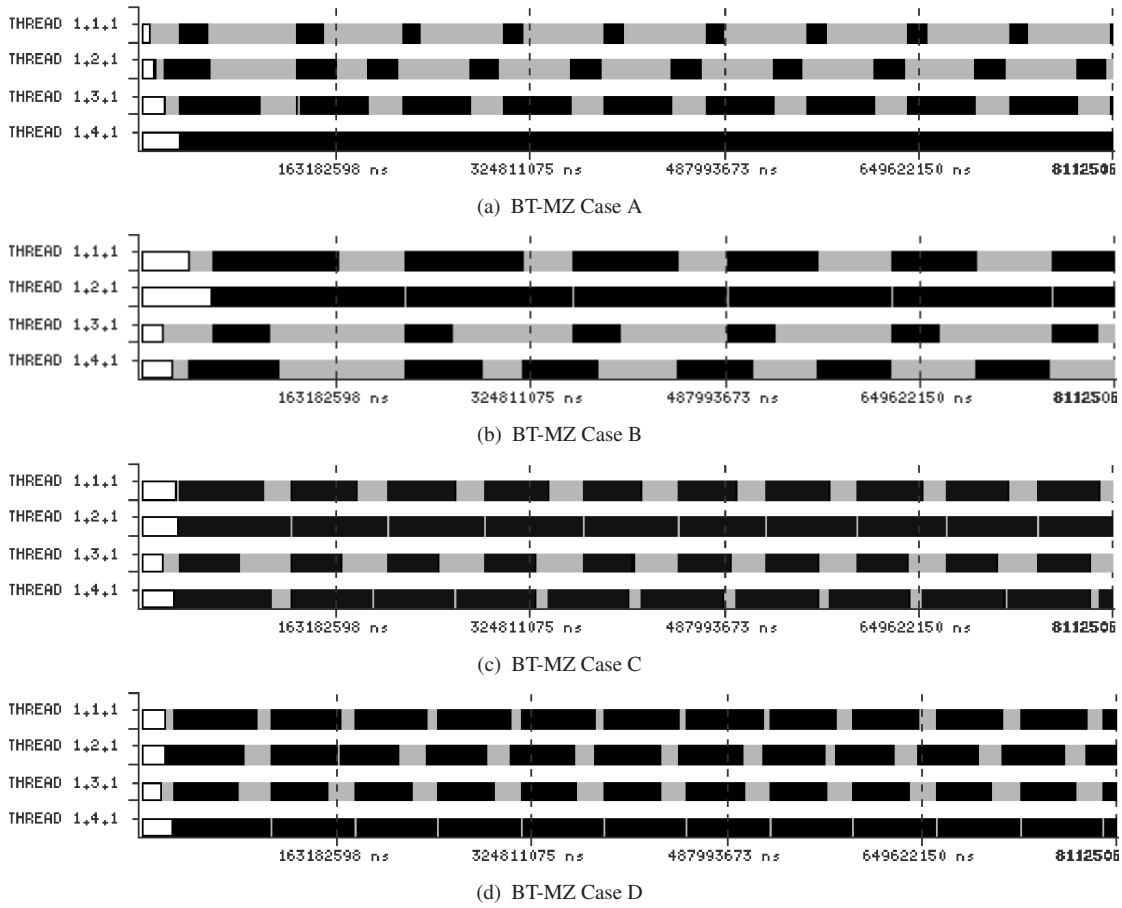
Test	Proc	Core	% Comp	Priority	Exec. Time
A	P1	1	24.32	4	81.64s
	P2	1	98.99	4	
	P3	2	24.31	4	
	P4	2	99.99	4	
B	P1	1	51.16	5	76.98s
	P2	1	99.82	6	
	P3	2	51.18	5	
	P4	2	99.98	6	
C	P1	1	98.96	4	74.90s
	P2	1	98.56	6	
	P3	2	97.01	4	
	P4	2	98.37	6	
D	P1	1	99.87	3	95.71s
	P2	1	73.25	6	
	P3	2	99.72	3	
	P4	2	73.25	6	

3.5.2 BT-MZ

Block Tri-diagonal (BT) is one of the NAS Parallel Benchmarks (NPB) suite. BT solves discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions, operating on a structured discretization mesh. BT Multi-Zone (BT-MZ) [48] is a variation of the BT benchmark which uses several mesh (named *zone*) for, in realistic applications, a single mesh is not enough to describe a complex domain.

Besides the complexity of the algorithm, BT-MZ shows a behavior very similar to our Metbench benchmark: every process in the MPI application performs some computation on its part of the data set and then exchanges data with its neighbors asynchronously (using `mpi_isend()` and `mpi_irecv()`); after this communication phase (which lasts for a very short time, around 0.10% of the total execution time) each process waits (with a `mpi_waitall()` function) for its neighbors to complete their communication phases. In this way, each process gets synchronized with its neighbors (note that this does not mean that each process gets synchronized with all the other processes). Once a process has exchanged all the data it had to exchange, a new iteration can start and the previous behavior repeats again till the end of the application (in our experiments we used BT-MZ with default values: class A with 200 iterations).

Case A: Figure 3.3(a) shows the BT behavior in the reference case, i.e. when process P_i is assigned to CPU_i and the priority of all the processes is 4. After an initialization phase (white bars at the beginning of the execution of each thread), all the processes



(a) BT-MZ Case A

(b) BT-MZ Case B

(c) BT-MZ Case C

(d) BT-MZ Case D

FIGURE 3.3: Effect of the proposed solution on BT-MZ. Each trace represents only some iterations of the application.

reach a barrier (synchronization point). From this point on, the real algorithm starts: during every iteration, each process alternate computing phases (dark) with synchronization phases (grey) at the end of communication phases (black).

It is easy to see from figure 3.3(a) that BT-MZ shows a great imbalance[3].

The imbalance is caused by the fact that some processes (for example process $P1$) have a small part of the data to work on, while other processes (for example, processes $P4$) have a large amount of data to take care of. It is also clear that process $P4$ is the bottleneck of the application and that speeding up this process will improve overall performance.

In order to solve the imbalance introduced by data repartition in BT-MZ, we ran process $P1$ and $P4$ on the same core and assigned more hardware resources to the latter, improving its performance while decreasing $P1$'s performance. This mapping should

[3]Even if the goal of this chapter is not to show whether SMT processors are useful in HPC or not, the table also shows the ST mode performance (only one process per core) of the application.

TABLE 3.2: BT-MZ balanced and imbalanced characterization

Test	Proc	Core	% Comp	Priority	Exec. Time
ST	P1	1	49.33	7	108.32s
	P2	2	99.46	7	
A	P1	1	17.63	4	81.64s
	P2	1	28.91	4	
	P3	2	66.47	4	
	P4	2	99.72	4	
B	P1	1	52.33	3	127.91s
	P2	2	99.64	3	
	P3	2	28.87	6	
	P4	1	46.26	6	
C	P1	1	65.32	4	75.62s
	P2	2	99.68	4	
	P3	2	53.78	6	
	P4	1	85.88	6	
D	P1	1	82.73	4	66.88s
	P2	2	73.68	4	
	P3	2	66.40	5	
	P4	1	99.72	6	

allow us to give a large amount of resources to process $P4$ without reversing the imbalance, i.e., without making process $P1$ slower than $P4$ like it was the case for Metbench (Case D). In fact, this mapping seems reasonable, for our goal is to increase the performance of $P4$ (the most computing intensive process) and we know that, with this operation, we will reduce the performance of the process running on the same core with $P4$. We chose $P1$ because it is the process with the shortest computation phase.

Case B: In our first attempt to reduce the imbalance we assigned priority 3 to processes $P1$ and $P2$ and priority 6 to processes $P3$ and $P4$. Figure 3.3(b) shows how 1) the imbalance has been inverted (process $P1$ now takes longer than $P4$ and 2) the new bottleneck is now process $P2$, which is also running with priority 3. As a consequence, the total execution time now takes longer (127.91 sec instead of 81.62 sec), which means the new bottleneck runs for much longer than the previous one.

Case C: In order to restore the original relative behavior between process $P1$ and $P4$ we incremented the resources assigned to process $P1$. Figure 3.3(c) shows that $P1$ now runs for less time than $P4$, as in Case A. As we can see, giving more resource to $P2$ (which is again the bottleneck) reduced the total execution time to 75.62 sec, with a 7.37% of improvement with respect to Case A.

Case D: Finally, we can argue that $P2$ and $P3$ execute their operation on a similar amount of data, therefore the amount of resources given to each process should not be as different as for $P1$ and $P4$. In our last test, we still assigned priority 4 to $P1$ and 6 to $P4$, as in the previous case, but we assigned priority 5 to $P2$ and 6 to $P3$, sharing resources between these two processes running on the same core more equally. Figure 3.3(d) shows that the imbalance has been reduced again with respect to Case C, in fact, now $P2$ and $P3$ compute more or less for the same amount of time. Also the new bottleneck is $P4$, which takes much shorter than $P2$ in Case C. Table 3.2 shows how the total execution time has also been reduced to 66.88 sec, with a 18.08% of improvement over the reference Case A.

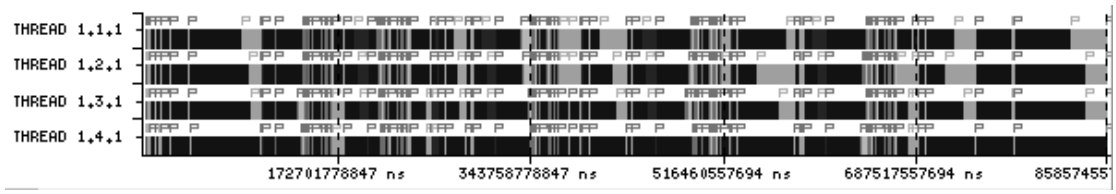
3.5.3 Siesta

Our last experiment consists of running SIESTA as an example of real application. SIESTA [72] is a method for *ab initio order-N materials simulation*, specifically it is a self-consistent density functional method that uses standard norm-conserving pseudo-potentials and a flexible, numerical linear combination of atomic orbitals basis set, which includes multiple-zeta and polarization orbitals.

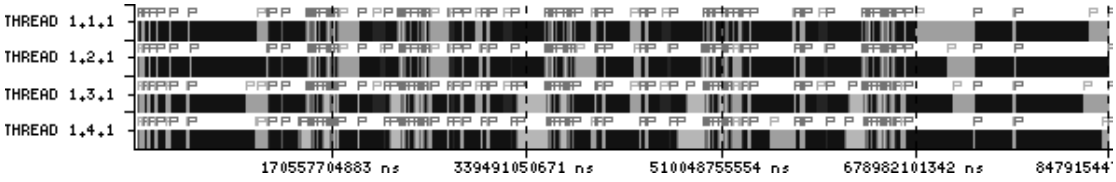
The application presents an imbalance caused by both the algorithm and the input set. SIESTA behavior, however, is not constant during each iteration, as can be seen in Figure 3.4(a); this makes our static balancing solution not as good as for the BT-MZ case. Yet, we achieved a improvement of 8.1% of execution time reduction with respect to the reference case (Case A).

Case A: Like for BT-MZ, Case A is the reference case, i.e., where process P_i is assigned to CPU_i and the priority of all the processes is set to 4. Figure 3.4(a) shows the trace for this reference case. The program starts with an initialization phase (11.99% of the total time) at the end of which each process in the application must reach a barrier. The initialization phase already presents some little imbalance, which evidences how the input set makes SIESTA not balanced. In the internal parts, each process exchanges data only with a subset of the other processes in the application, and then reaches a synchronization point (`WaitAll()`), waiting for all the others to complete their jobs. In the last part, the processes finalize their work (13.41% of the total time): after the last barrier, each process computes its function on its sub-set of data and then ends. A complete execution of the program in this configuration takes 858.57 secs.

Case B: As we can see from the trace in Figure 3.4(a) is not easy to understand how to balance the application and whether our balancing approach is worth. However,

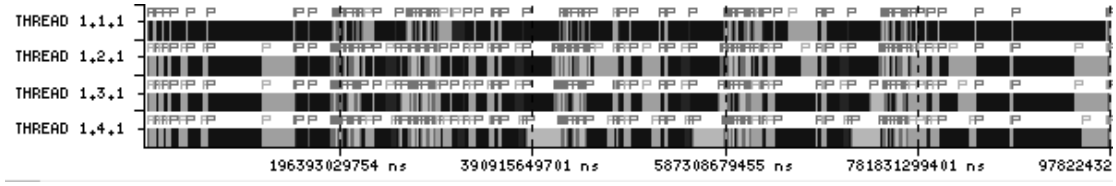


(a) SIESTA Case A



(b) SIESTA Case B

(c) SIESTA Case C



(d) SIESTA Case D

FIGURE 3.4: Effect of the proposed solution on SIESTA.

Table 3.3 shows some more information about SIESTA (hard to retrieve from the trace): processes $P1$ and $P2$ spend a considerable amount of time waiting for $P3$ and $P4$ to reach the barrier. Thus, the first hint would be to put $P1$ and $P3$ on one core and $P2$ and $P4$ on the other and then play with priority. We tried this case but then we realized that $P2$ and $P3$ have almost the same amount of data to work on. Thus, in Case B we put $P2$ and $P3$ on the first core and $P1$ and $P4$ on the second one and increased the priority of $P3$ and $P4$ to 5. In this case we achieved a little improvement of 1.24% (the total execution time is 847.91 sec). Figure 3.4(b) shows that, in this new configuration, $P2$ is the new bottleneck of the finalization part.

Case C: In the previous case we obtained a little improvement, still the application results quite imbalanced. We realized that, since $P2$ and $P3$ work, more or less, on the same amount of data, using a different priority for these two processes may introduce even more imbalance. Figure 3.4(b) shows that, indeed, this is the case. In Case C we restored the original relative behavior between process $P2$ and $P3$ setting both their priority to 4 (i.e., the difference is 0). Figure 3.4(c) shows how the application is now more

TABLE 3.3: SIESTA balanced and imbalanced characterization

Test	Proc	Core	% Comp	Priority	Exec. Time
ST	P1	1	81.79	7	1236.05s
	P2	2	93.72	7	
A	P1	1	75.94	4	858.57s
	P2	1	75.24	4	
	P3	2	82.08	4	
	P4	2	93.47	4	
B	P1	2	79.57	4	847.91s
	P2	1	87.06	4	
	P3	1	72.04	5	
	P4	2	77.73	5	
C	P1	2	83.04	4	789.20s
	P2	1	79.66	4	
	P3	1	80.78	4	
	P4	2	78.74	5	
D	P1	2	90.76	4	976.35s
	P2	1	65.74	4	
	P3	1	68.08	4	
	P4	2	63.95	6	

balanced. For example, looking at the initialization and the finalization part, it is possible to see that the processes are much more balanced than in Case A and Case B. In fact, re-balancing SIESTA reduces the total execution time to 798.20 sec, an improvement of 8.1% with respect to the reference case.

Case D: Following the same idea of the previous case (i.e., leave $P2$ and $P3$ with the same priority and play with $P1$ and $P4$), we increased the amount of resources assigned to $P4$, penalizing $P1$. Figure 3.4(d) shows how we reverse the imbalance: SIESTA is again imbalanced, though in a different way than in the reference case. In Case D, $P1$ (the process with less hardware resources) is the bottleneck (in the initialization, finalization and most of the internal phases) and the total execution time increases to 976.35 sec, with a loss of 13.72%.

BT-MZ and SIESTA are two cases of non-balanced HPC applications, though their imbalance is quite different. BT-MZ executes several iterations, all of them similar from the execution time, CPU utilization and imbalance point of view. SIESTA also executes several iterations but each iteration is not necessarily similar to the previous or the next one. In particular, the process that computes the most is not the same across all the iterations. For example, in the *i*-th iteration $P1$ could be the bottleneck while in the *(i+1)*-th the most computing process could be $P4$. This behavior suggests that a good balancing mechanism would prioritize $P1$ in the *i*-th and $P4$ in the *i+1*-th iteration.

Our static approach does not allow us to play in this way as we assign the priority at the beginning of the execution and never change them during the execution. We argue that a dynamic mechanism is required to correctly set priorities for applications that change their behavior throughout their execution. Since real applications are likely to behave like SIESTA rather than like BT-MZ, we intend to extend our balancing mechanism as part of the Operating System, so that the OS can dynamically set the priority of each process according to actual application behavior.

3.6 Conclusions

In this chapter we showed how allowing software to control the amount of shared resources assigned to each thread in a MT processor may improve the performance of HPC applications. In fact, some applications show an imbalanced behavior, i.e., some processes require more time to complete their computing phase while all the other processes are waiting at some synchronization point and cannot move forward. While the imbalance can be caused by either external or internal factors (most likely both), it is clear that it may reduce the performance of an HPC application, resulting in a significant waste of resources in Supercomputers. Our results show how using our modified Linux kernel to control a processor capable to dynamically assign processor resources to running threads (the IBM POWER5 in our case), reduces the application imbalance and, therefore, improves overall performance. The experiments we performed show an improvement up to 18% for a widely used BT-MZ benchmark and up to 8.1% for a real application. We achieved these results without putting the burden of balancing the application on the programmer and regardless of the used programming model.

Our results suggest that an automatic mechanism could even increase the actual improvement, thus, motivating the use of MT processors with the capability to re-assign hardware resources between threads in future Supercomputers. In the next chapter, we extend our OS by introducing an algorithm able to automatically detect if a process deserves a higher amount of resources and which process should be deprived of those resources so that imbalance can be reduced. In addition, a user-level mechanism is proposed in Chapter 5.

Chapter 4

A Dynamic Scheduler for Balancing HPC Applications

4.1 Introduction

Modern Supercomputers are often designed with commodity hardware components (for example, Intel or IBM POWER processors) and software. Generally, this kind of Supercomputers are distributed memory machines with a limited number of cores per-node (2-8 cores); the Message Passing Interface (MPI) [3] standard is the most common programming model used in those systems.

In Chapter 2 we performed a deep analysis of how the hardware prioritization mechanism of POWER5™ processors affects the performance of applications. Two of the main conclusions, also used in this chapter, are the following:

1) In general, improving the performance of one task involves a higher performance loss on the task running on the other context, sometimes by an order of magnitude. In some cases, in order to reduce the execution time of a task by X% (with respect to the case when both tasks run with the same priority) by increasing its priority, the execution time of the other task in the same core may reduce by more 10X%.

2) Instead of using the full spectrum of priorities (from 0 to 7), we only explore priority differences up to ± 2. Larger priority differences should be used mainly when the performance of one of the two tasks is not important (e.g., background task).

In the previous chapter we showed how the hardware prioritization mechanism of POWER5 processors can be used to balance HPC applications. As a proof-of-concept, we ran a

4-tasks MPI application on a POWER5: in a first test, where we applied the same priority to the two tasks running in a core (default case), we detected which processes, on average, computed the longer and which tasks spent most of their time waiting for incoming messages or on a barrier. In the following experiments we manually increased the priority of the most computing intensive tasks, increasing their speed and reducing the load imbalance. In that chapter, the prioritization is applied to processes manually and statically at the beginning of the execution and each process runs with the same priority throughout its execution. With this solution we obtained an improvement of 8% on real HPC applications like SIESTA [2].

In the current chapter, we propose a dynamic solution implemented as a new task scheduler for Linux 2.6 kernels. The advantages of this new proposal over the static solution are obvious, the most important being that the OS automatically establishes the hardware priority to be assigned to each HPC process with no effort from the user. The second advantage is that the solution is transparent to the user: the only modification in the application source code concerns the scheduling policy (as shown in Section 4.3). The third advantage is that our scheduler is able to detect the correct hardware priority quickly (in one or two iterations) improving overall performance. Finally, the scheduler is able to catch up with the application in case the application's behavior is dynamic, i.e., not constant throughout the iterations. All these advantages reduce the load imbalance of a HPC application, directly increasing the overall performance.

In order to test our dynamic scheduler, we compared the results we obtained running HPC benchmarks and applications to the results we obtained in the last chapter. Most interesting is the case of the real application (SIESTA): with our previous static approach we were able to improve the total execution time by 8%; with the solution proposed in this chapter, we are able to improve the execution time by almost 6%, combining the effects of the load balancing and the high-responsive task scheduler without any effort from the programmer.

The capability of the IBM POWER5 to allow the software to change processor's internal resource allocation is not something isolated in the design of processors. Several factors support the idea that future supercomputers will use this type of processors. First, nowadays, Multi-Threaded processors are widely used in HPC systems (in addition to many other computing systems like desktops, real-time, etc.) for their good performance/energy consumption and performance/cost ratios. Second, other recent processors like the IBM POWER6^{TM} [53], provide a similar prioritization mechanism. Third, many computer-architecture researchers advocate that allowing the software to control not only the decode stage of the processor, as it is the case in POWER5 and

POWER6, but also other processor shared resources in the chip, like the cache [40, 46], would increase the performance of HPC applications.

The rest of this chapter is structured as follows: Section 4.2 highlights some of the features of the software designs of the new Linux scheduler framework. Section 4.3 proposes our dynamic task scheduler for balancing HPC applications. Section 4.4 shows our experiments on benchmarks and real applications. Finally Section 4.5 provides our conclusions and finalizes the paper.

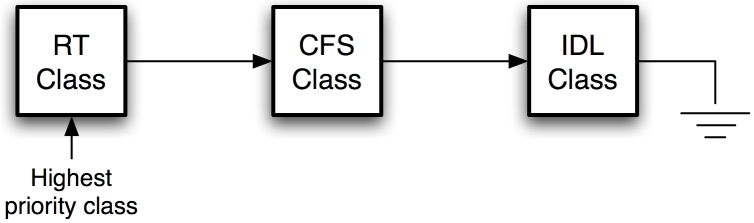
4.2 The Linux Scheduler Framework

A new process task scheduler (the *Complete Fair Scheduler*, CFS) has been introduced in the Linux kernel version 2.6.23. This new scheduler replaces the old *O(1)* [10] scheduler used in Linux 2.6 for several years. The *O(1)* scheduler provided good performance and its overhead was constant regardless of the number of runnable processes. However, this scheduler was not free of problems, such as consuming too much memory even with few runnable tasks. The CFS aims to solving some of those problems.

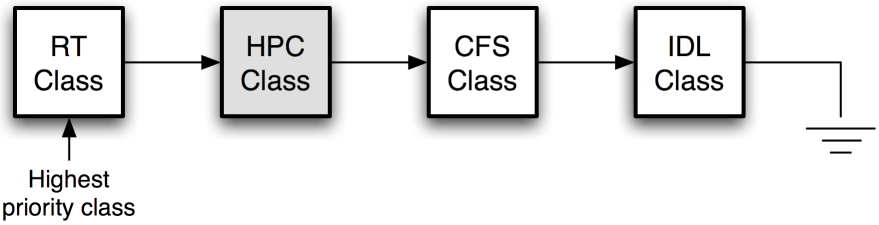
Together with the new CFS algorithm, a new *scheduler framework* has also been introduced, mainly to simplify the structure of the task scheduler. The new framework divides the scheduler in two main components: three *Scheduling Classes*, which implement the policy details, and a *Scheduler Core*, which handles the Scheduling Classes as *objects*, i.e., calling the appropriate Scheduling Classes methods for any low-level operations (for example, selecting the next task to run or accounting for the time elapsed). Each of the three Scheduling Classes contains one ore more scheduling policies (see Figure 4.1(a)).

In order to improve scalability, each CPU has a list of Scheduling Classes. Each class, in turn, contains a list of runnable processes belonging to one of the policies handled by the class. The first class (the highest priority) contains real-time processes (`SCHED_FIFO` and `SCHED_RR`); the second class (the new CFS class) contains the normal processes (`SCHED_NORMAL`, previously called `SCHED_OTHER`, and `SCHED_BATCH`); finally, the last class contains the idle process (`SCHED_IDLE`).

The order with which the Scheduling Classes are linked together introduces an implicit level of prioritization: no processes from a low priority class will be selected as long as there are available processes in one of the higher priority classes. For example, no processes from the CFS class will be selected if there is one process in the real-time



(a) Standard Linux Scheduling Classes



(b) HPCSched Scheduling Classes

FIGURE 4.1: Scheduling classes for the standard and the modified Linux kernel

class; this design choice preserves the semantic of the `SCHED_FIFO` and `SCHED_RR` policies. In the same way, the idle process will never be selected if there are runnable processes in one of the other classes.

When the scheduler is invoked, the Scheduler Core starts looking for the best process to run from the highest priority class (i.e., the real-time class) and checks whether there are runnable processes in this class. If the class contains at least one process, the scheduler selects this process and assigns it to the CPU. If the class is empty, i.e., no runnable process available, then the Scheduler Core moves to the next class. This operation repeats until the Core Scheduler finds a runnable task to run on the CPU. Notice that the idle class always contains at least the idle process, thus the scheduler cannot fail in its search.

A very interesting property of the new scheduler framework is that each class may provide different data structures and algorithms to select the next process to run. For example, the real-time class uses a set of priority, round-robin run queue lists, one list for each real time priority (0-99). The real-time scheduler first selects the highest (non-empty) priority run queue and then picks up the first task in the list. In fact, a real-time task is either `SCHED_FIFO`, in which case the task stays in the first position until it yields the CPU, or a `SCHED_RR`, in which case the process is moved to the back of the queue if its time slice expires. This algorithm is essentially the old *O(1)* scheduler algorithm and maintains the *O(1)* scheduler's implementation details (like the 0-cost swap between the active and expired arrays).

The CFS class, instead, uses a red-black tree and does not use the concept of time quantum. Each process receives a time slice proportional to the actual workload (the higher the number of running processes, the smaller the time slice). The key concept is the time spent by a runnable task waiting for a CPU (i.e., waiting to be executed). This value is used to sort the tasks in the red-black tree so that the "leftmost task" in the tree is the process that has been waiting for more time (i.e., the one with gravest need to run), therefore the next task to run. The CFS scheduler tries to balance the execution of the runnable tasks so that no one waits for a CPU more than a maximum allowed amount of time[1] (*latency*). As the time passes, the waiting time of the running process is decreased at every timer interrupt (or scheduling event) by the amount of time the task has been running (minus its fair running time). As the waiting time of the running task decreases, the task may eventually be moved to the right side of the red-black tree. Sooner or later the running task will not be the "leftmost task" anymore, in that moment the CFS scheduler will select another task.

As the previous examples show, the Scheduling Classes may have completely different algorithm and data structures. As a matter of fact, the new scheduler framework allows kernel developers to write scheduler algorithms specifically tailored for a class of applications. Moreover, adding a new scheduler algorithm is easier than in the past and does not require heavy modification of pre-existing kernel code.

4.3 The HPC Scheduler

In this chapter we propose a dynamic mechanism to balance MPI applications using the hardware priority mechanism provided by IBM POWER5 processors. We implemented our dynamic solution inside the Linux kernel as a new scheduler (*HPCSched*) for a special class of applications (HPC applications).

In order to balance the HPC application, the scheduler tracks the application behavior and detects when to increase or decrease the amount of processor's internal resources assigned to a specific process.

Since we want to prioritize HPC over normal processes, we introduced the HPCSched class between the Real-Time and the CFS class (see Figure 4.1(b)). In this way, we preserve the semantic of the real-time tasks (`SCHED_FIFO` and `SCHED_RR`) and give a higher priority to HPC processes over normal tasks.

[1] The default maximum value for normal tasks is 20ms.

The HPC scheduler we propose is based on three components, mainly independent from each other:

Scheduling policy: The scheduler algorithm used by the Scheduler Core to select the next task to run among the runnable tasks in the HPC class.

Load Imbalance Detector and Heuristics: We use a Load Imbalance Detector and heuristic functions to select, according to the scheduler metrics, the new hardware priority for the task.

Mechanism: Architecture-dependent, utility functions necessary to set the new hardware priority or read the current priority of a task.

4.3.1 Scheduling policy

Taking advantage of the new scheduler framework described in the last Section, we introduced a new Scheduler Class (`sched_hpc`) and a new scheduler policy (`SCHED_HPC`) for HPC applications. A user can move an application to the HPC class by means of the standard `sched_setscheduler()` system call. Actually, this is all the effort the user has to put in order to use our new scheduler (comparable to the use of the `nice()` system call commonly used in HPC applications).

Our scheduler algorithm is specific for HPC applications, more specifically for MPI applications. The typical way of running MPI applications on current supercomputers is to run one MPI process per-CPU. Thus, we expect to have one process in the HPC class of every CPU (maybe two or three during workload balancing). Under this assumption, it is not worth to have a complex algorithm for selecting the next task to run. In fact, with this small number of processes in the run queue list, a simple round-robin list is as good as a more complex red-black tree. However, the code for a round-robin run queue is much simpler and performing (for example, the scheduler does not have to balance any tree). Nevertheless, we implemented two algorithms:

FIFO: Fist-In-First-Out algorithm. The selected task will run until the end or until it yields the CPU.

RR: Round-Robin algorithm. Each task has a pre-defined time slice. When this time slice expires, the task is placed at the end of the run queue.

We observed that, with one process per CPU running at any given moment, there is essentially no difference between these two policies, thus, we only include the results

for the round robin policy in this chapter. However, as we have already remarked, the scheduling policy is independent of the other components, hence, it can be changed, if required, without affecting the heuristics or the applying mechanism.

In the new Linux kernel framework, workload balancing, i.e., splitting evenly the workload among all the available domains [10] (at core-, chip- and system-level), is also performed at Scheduling Class level. Every Scheduling Class has its own workload balancing algorithm, which means that each CPU has, roughly, the same number of real-time or normal tasks. As a side effect, each CPU runs, more or less, the same number of tasks.

The workload balancer is invoked whenever the kernel detects that there is a big imbalance or if one processor is idle. In the latter case, the idle CPU tries to pull tasks from other, busiest run queue lists to its run queue.

We implemented our HPC workload balancing algorithm making each processor domain [10] running the same number of processes. For example, in a POWER5 system there are three domain levels: chip level, core level and context level (a context is what is recognized by the OS as a CPU). Our workload balancer tries to balance the number of task at each domain level. Thus, a core domain running less tasks than another core will try to pull tasks from the other core. For example, if one core of an IBM POWER5 processor (a domain composed by two contexts) contains one HPC task and the second core contains three tasks, the first core will try to pull one HPC task from the second core so that each core domain contains two processes so to make the workload balanced.

4.3.2 Load Imbalance Detector and Heuristics

MPI applications alternate *computing phase* (when a process is runnable) with *waiting phases* (when a process is waiting for an incoming message or for synchronization, thus, not runnable). We consider the sum of a computing phase and of a waiting phase as one *iteration* of the MPI application.

In some HPC applications during each iteration all the tasks perform the same operations (most of the time on the same amount of data), with an iterative structure.

Our solution learns from the execution history of a process: the general idea is that if a task does not have a high CPU utilization during the iteration i, it will perform in the same way in the $i + 1$ iteration. This is a common case, for example, for those applications that compute an approximation of a solution of a problem and than try

to reduce the error in they made in the approximation. The Load Imbalance Detector assumes that the iteration i is representative of the iteration $i + 1$, hence, the HPC scheduler can change the task's priority and apply the new priority before the iteration $i + 1$ starts. The goodness of our solution strongly depends on how close this guessing is to the optimum solution. If the guessing is not correct, in the iteration $i + 1$ the application may result to be even more imbalanced than in the iteration i. Hopefully, the scheduler will detect this anomaly during the iteration $i + 1$ and apply the right priority in the iteration $i + 2$.

Clearly, not all the applications present a well defined iterative structure with a barrier at the end of the iterations. Some applications, like SIESTA, are more dynamic or do not require all the processes to be synchronized with a global barrier. If the iteration i is not representative of the iteration $i + 1$, our current heuristics will probably fail to balance the application and new heuristics are required. We leave the study of new heuristics for future work.

The scheduler may require some iteration to converge to a balanced solution: the goal of the heuristic is to find a stable state where the application is balanced and to remain there as long as the application behavior is constant. Sometimes it is not possible to balance an application, for example because the hardware priority mechanism of the POWER5 processor is too coarse grain. In this case the scheduler will oscillate between two solutions without being able to find the perfect balance, hopefully still reducing the overall load imbalance.

The problem here is to find the correct trade-off between *performance* (computing the next priority quickly), *responsiveness* (converging to the correct priority in few iterations) and *adaptability* (changing the priority whenever the tasks' behavior changes).

In order to compute the next task priority quickly our heuristics are based on the CPU utilization of a process, a simple metric that does not require complex computations. Ideally, the scheduler should look at the tasks running on the two contexts of a POWER5 core simultaneously and then compute the correct priority for the current task. In fact, the performance of the current task depends on the difference between its priority and the priority of the task running on the other context. However, this would require to acquire a lock on the other context's run queue (in order to ensure that no process switch occurs), thus, stalling the other context until the new priority has been computed. Things become even more complex as the HPC scheduler needs to be sure that the process running on the other context is a `SCHED_HPC` tasks, for the lock on the task descriptor should also be acquired (in order to avoid concurrent access to the task descriptor). This solution could be quite expensive in terms of performance (though very precise).

Hence, we decided to implement a simpler solution that only computes the new priority of a HPC task according to its statistics (thus, not considering the task running on the other context).

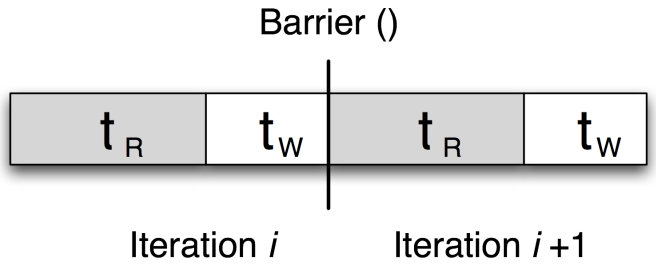


FIGURE 4.2: HPC application iterative behavior

While a task is running, the scheduler collects several metrics, such as the tasks' execution and waiting time. Figure 4.2 shows a typical task trace: the process computes for t_R seconds and then goes to sleep, waiting for messages coming from the other processes in the MPI application (t_W). If $t_i = t_R + t_W$ is the total execution time in the iteration i, then the task utilization in the same iteration is $U_i = t_R/t_i$. The global task utilization is the ratio of the accumulated running and iteration times: $U = \sum t_R / \sum t_i$. These metrics are quite easy to compute, since the kernel already provides some of the required values. We only had to add the values necessary to introduce the concept of *iteration* that is not present in the standard Linux kernel.

From our study in [8], we learned that priority differences greater than 2 drastically reduce the performance of the low priority task. Therefore, we limited the range of priorities that the HPC scheduler explores to $[4, 6]$ (where 4 is the normal priority assigned to each task at the beginning), so that the maximum allowed priority difference is ± 2. In this way, the performance of the highest priority task might increase up to 95% of the maximum performance improvement but the lower priority task's performance does not decrease too much.

Once the information about the tasks' progress have been stored, the HPC scheduler has to decide whether to increase, decrease or keep the same priority for the current process in the next iteration. Since HPC applications can be very different, it is hard to find an heuristic that works well in all the cases. In this chapter, we implemented and tested two heuristics: the first heuristic (*Uniform* heuristic) targets constant applications, i.e., applications that do not change drastically their behavior from one iteration to another. The second heuristic (*Adaptive* heuristic) is more aggressive and tries do adapt to different program phases. Which heuristic is better for a specific application depends on the characteristics of the applications itself. Section 4.4 shows how an application takes

more advantages from one heuristic than from the other. We decided to allow the user to select which heuristic to use when compiling the kernel. Once the heuristic has been chosen, the user can set some parameters at run time to tune the heuristic and make it more suitable for the application.

Uniform prioritization: This heuristic uses the global utilization ratio of a task. Every scheduling tick, the OS accumulates the running time for the active task and updates its utilization; the sleeping time is accounted when a task wakes up at the beginning of the new iteration. Just before starting the new iteration, the Load Balancer Detector checks the application's imbalance and the heuristic eventually applies the new task priority according to the global utilization,

We introduced two configurable limits, `LOW_UTIL` and `HIGH_UTIL` that define the boundaries when a task is considered to be a low, medium or high utilization task. Those boundaries are required to avoid that the scheduler changes too quickly the priority of a task, oscillating between two possible solutions. For the experiments presented in Section 4.4, we set `HIGH_UTIL` to 85 and `LOW_UTIL` to 65. The heuristic can be tuned by the user through specific entries in the *sysfs* filesystem.

The Uniform heuristic is very simple and adds negligible overhead to the task scheduler. The heuristic properly balance applications with constant behavior although it could be slow to adapt to different behaviors of the program. If the heuristic is able to balance the application, i.e., to find a *stable state*, the Load Imbalance Detector only checks whether the application maintain the same behavior or not, without changing the priority of each task. If the application's behavior changes, the Load Imbalance Detector tracks this and the heuristic selects the right priority for the next iterations.

Adaptive prioritization: The *Uniform* heuristic may be too slow to adapt to new scenario if the application changes its behavior quickly, especially if the application runs for a long time (in which case it is hard to impact the global utilization, as Section 4.4.2 shows. We implemented another heuristic, that we called *Uniform*, which gives more weight to the recent history of the application. With this heuristic, the task utilization in the $i-th$ iteration is computed as $U_i = G * U_g(i-1) + L * U_l(i)$, where $U_g(i-1)$ is the global utilization until the iteration $i-1$ and $U_l(i)$ is the CPU utilization of the last iteration i. G and L (with $G + L = 1$) weight, respectively, the global and the last utilization. These parameters can be used to make the heuristic more or less aggressive: in fact, an aggressive heuristic (for example, $L = 0.90$ and $G = 0.10$) quickly adapts to the application's behavior but may over-react, meaning that even small changes caused

by external factors, like the OS noise, may cause the heuristic to change the task priority. On the other hand, if the value of G is close to 1, the *Adaptive* heuristic behaves like the *Uniform* heuristic.

As for the Uniform heuristic, the Adaptive heuristic can also be tuned at run time using different values for `HIGH_UTIL`, `MAX_PRIO` (the maximum allowed priority) and `MIN_PRIO`. Moreover, if the Load Balancer stops to change the tasks' priority if it detects that the application is well balanced.

4.3.3 Mechanism

This is the only architecture-depended part of our solution. In fact, while the HPC scheduler can be used on any architecture and may, eventually, provide some performance improvement (because the HPC class has higher priority than the CFS class), balancing an MPI application assigning more or less hardware resources to a process can only be done if the underneath processor supports this feature.

4.4 Experiments

In this section we evaluate the performance of our HPC scheduler and compare it to the standard CFS scheduler and the static solution proposed the previous chapter.As we said in Section 4.3, the goodness of the HPC scheduler strongly depends on the heuristics we apply. For this reason, some application may benefit more than other from an heuristic while other may experiment some performance degradation.

Like in the previous chapter, we present our results for three different cases: Metbench, our micro-benchmark suite (Section 4.4.1), BT-MZ from the NAS benchmark suite (Section 4.4.3) and SIESTA, a real application (Section 4.4.4). In order to evaluate how our HPC scheduler handles dynamic applications, in this chapter we also present results for MetbenchVar (Section 4.4.2), a version of Metbench that changes its behavior after k iterations, reversing the load imbalance.

We performed the experiments on an IBM OpenPower 710 server, equipped with one POWER5 processor. We ran our experiments on a standard Linux 2.6.24 (the last available Linux kernel at the moment of writing this paper) and our modified Linux kernel, also based on the same Linux kernel version. All the benchmarks are MPI applications (in the experiments we used the MPI-CH 1.0.4p1 implementation of MPI protocol).

In order to graphically show how HPCSched balances an MPI application, we used PARAVER [52], a visualization and performance analysis tool developed at CEPBA to collect data and statistics and to show the trace of each process during the tests.

As a performance metric we use CPU utilization of each task, and the total execution time of the application. Reducing the load imbalance lead to higher CPU utilization but does not necessarily improve performance: other factors, like the communication pattern of the application, may play an important role and reduce the performance of the application. On the other hand, HPCSched is also able to improve the performance of an application reducing the overhead an application running with the standard CFS scheduler may suffer.

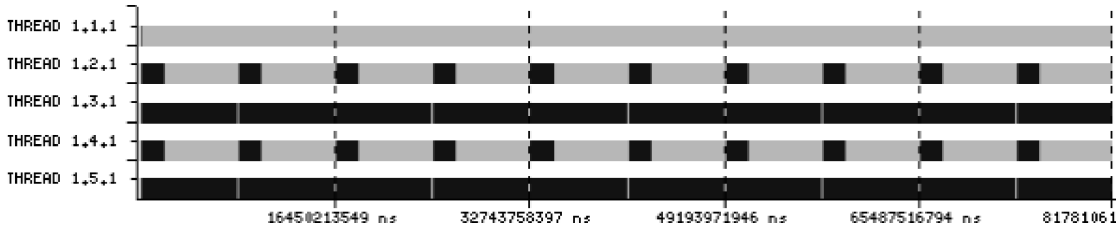
4.4.1 Metbench

Metbench (Minimum Execution Time Benchmark) is a suite of MPI micro-benchmarks developed at BSC which structure is representative of the real applications running on MareNostrum. It is described in Chapter 3, Section 3.5.1.

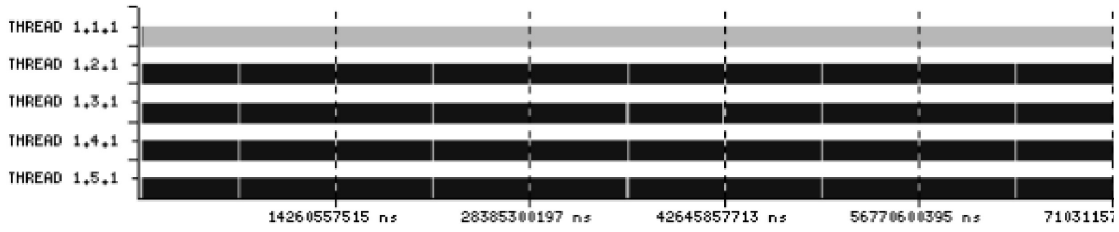
Figure 4.3(a) shows part of the execution trace of our reference case where Metbench runs with the default CFS (Completely Fair Scheduler). In this figure, dark areas represent the computing time, while the gray show the waiting or communication time. Table 4.1 shows that two of the Metbench workers are idle for about 75% of the time. Figure 4.3(b) shows the solution proposed in [9], where we were able to statically balance the application: the execution time decrease from $74.64sec$ to $70.90sec$, with an improvement of about 13%. The static approach we used in [9] require previous knowledge of the application and effort from the programmer to detect the load imbalance and to properly assign hardware resources to each task.

Figures 4.3(c) and 4.3(d) show how HPCSched is able to properly balance Metbench after the first iteration. In fact, the behavior of Metbench is constant, thus, each iteration is representative of the following ones. In Figure 4.3(c), the Load Imbalance Detector detects the imbalance in the first iteration [2] and the *Uniform* heuristic computes and apply the correct priority for each task before the beginning of the second iteration. At the end of the second iteration, the Load Imbalance Detector detects no imbalance, thus there is no need of trying to balance again the application. The execution time with the *Uniform* heuristic is $71.74sec$ (about 12% of improvement), comparable with the static solution shown in Figure 4.3(b) but without any effort from the programmer.

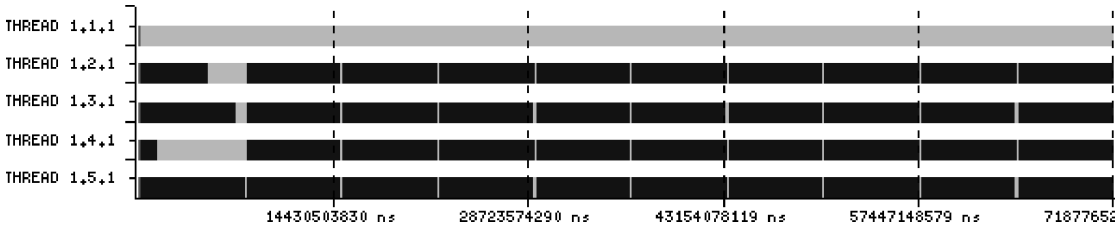
[2] Notice that the first iteration already uses non standard priority: this is the result of the initialization phase, not visible in the trace



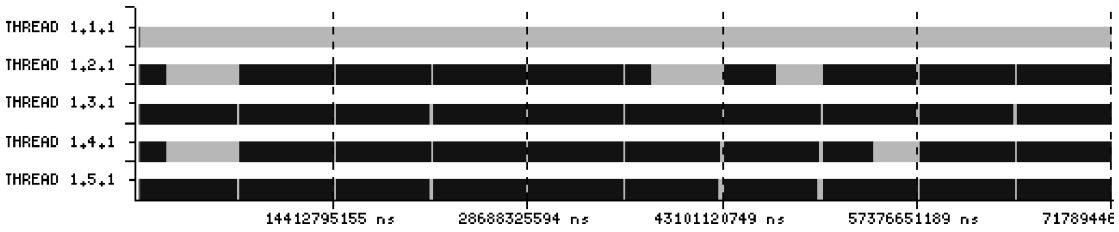
(a) Standard execution



(b) Static prioritization



(c) Uniform prioritization



(d) Adaptive prioritization

FIGURE 4.3: Effect of the proposed solution on Metbench.

The *Adaptive* heuristic also provides good performance: the total execution time is $71.65sec$ (about 12% of improvement). In this experiment the *Adaptive* heuristic uses a very aggressive approach (10% global history, 90% last iteration), thus, even a small variation (caused, for example, by OS noise) may stimulate the heuristic to change the priority of some task. If this happens, like in Figures 4.3(d), the heuristic may respond too quickly and take the wrong decision. However, Figures 4.3(d) also shows how the *Adaptive* heuristic is able to recover after the error.

4.4.2 MetbenchVar

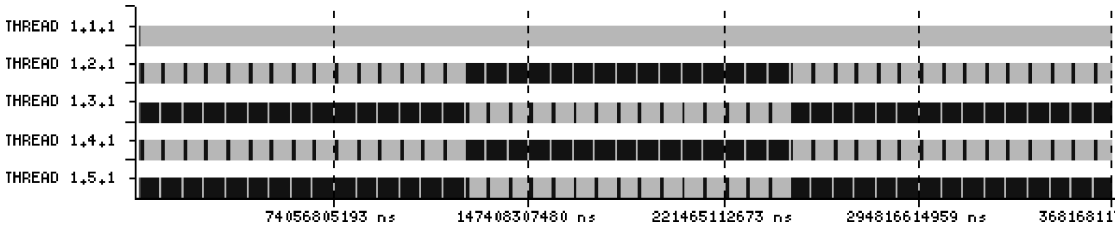
MetbenchVar is a slightly modified version of Metbench where the workers change their behavior after k iteration. Figure 4.4(a) shows the standard execution of MetbenchVar

TABLE 4.1: Metbench balanced and imbalanced characterization

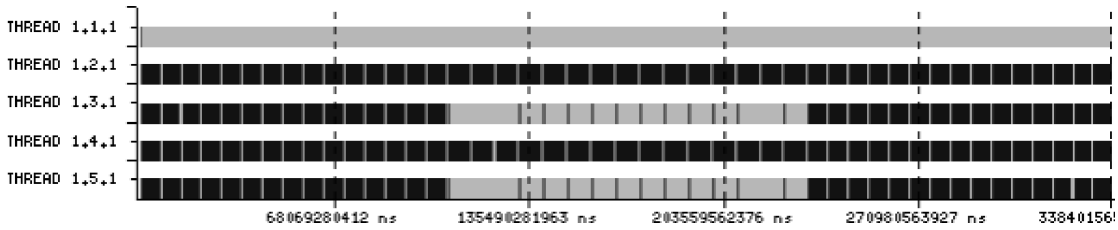
Test	**Proc**	**% Comp**	**Priority**	**Exec. Time**
Baseline	P1	25.34	4	81.78s
2.6.24	P2	99.98	4	
	P3	25.32	4	
	P4	99.97	4	
Static	P1	99.97	4	70.90s
	P2	99.64	6	
	P3	99.95	4	
	P4	99.64	6	
Uniform	P1	96.17	-	71.74s
	P2	98.57	-	
	P3	90.94	-	
	P4	99.57	-	
Adaptive	P1	80.64	-	71.65s
	P2	99.52	-	
	P3	87.52	-	
	P4	99.20	-	

with $k = 15$: at the beginning $P1$ and $P3$ execute a small load while $P2$ and $P4$ a large load. At the 15th iteration, $P1$ and $P3$ start to execute the large load while $P2$ and $P4$ perform their task on the small load. In this way, we reverse the load imbalance at run time making the application's behavior dynamic. At the 30th iteration, we switch again the behavior of the tasks. Figure 4.4(b) shows how a static works in this case: the application is perfectly balanced in the first (iterations 1-15) and third period (iteration 31-45) but the prioritization is reversed in the second period (iterations 16-30), as a result, in the second period the application performs worst than in the standard case.

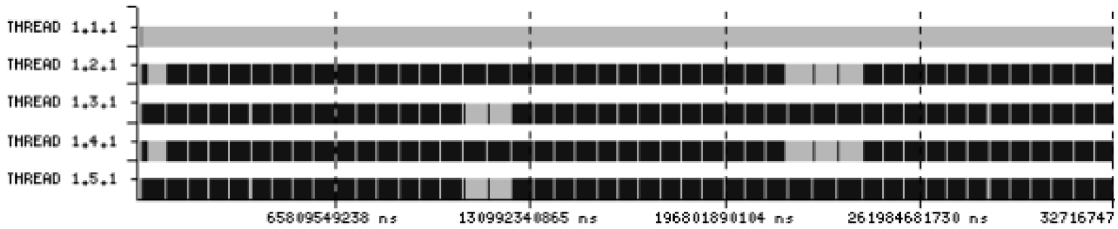
Our dynamic solution, instead, is able to detect that the application's behavior has changed and dynamically adjust the priority of each task in order to re-balance the application. Figure 4.4(c) shows how HPCSched performs in this experiment when applying the *Uniform* heuristic: after the switching in the 15th iteration, the scheduler needs two more iterations to detect and correct the new load imbalance. However, after the second switch, the scheduler needs three more iterations to detect and correct the load imbalance and the trend continue if the application runs for longer time. Since the *Uniform* heuristic uses the global history to detect the imbalance, it is expected that the longer the application runs, the less responsive is the scheduler. Thus, increasing the value of k or the number of periods makes the scheduler slower to adapt to the new scenario. As Table 4.2, the execution time reduces from $368.17sec$ to $327.17sec$, with an improvement of about 11%).



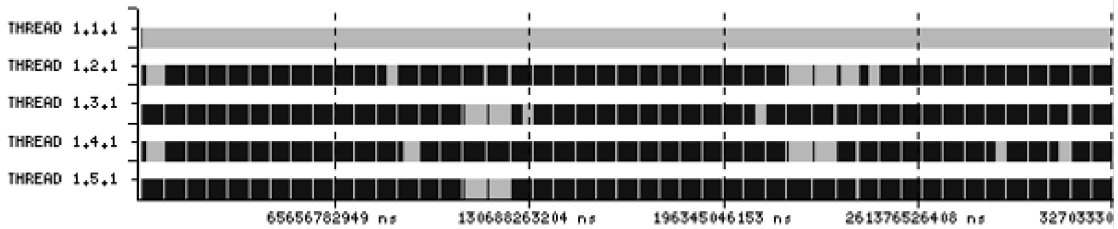
(a) Standard execution



(b) Static prioritization



(c) Uniform prioritization



(d) Adaptive prioritization

FIGURE 4.4: Effect of the proposed solution on MetbenchVar.

Figure 4.4(d) shows how the *Adaptive* heuristic preforms in this experiment: with $k = 15$, the scheduler always needs only two iterations to detect and correct the load imbalance but, as for the previous case, some times the heuristic is too aggressive and respond too quickly. Again, the *Adaptive* heuristic is able to correct its over-reaction in the following iteration and to reduce the execution time to $326.41sec$ (about 11% of improvement).

TABLE 4.2: Variable-Metbench balanced and imbalanced characterization

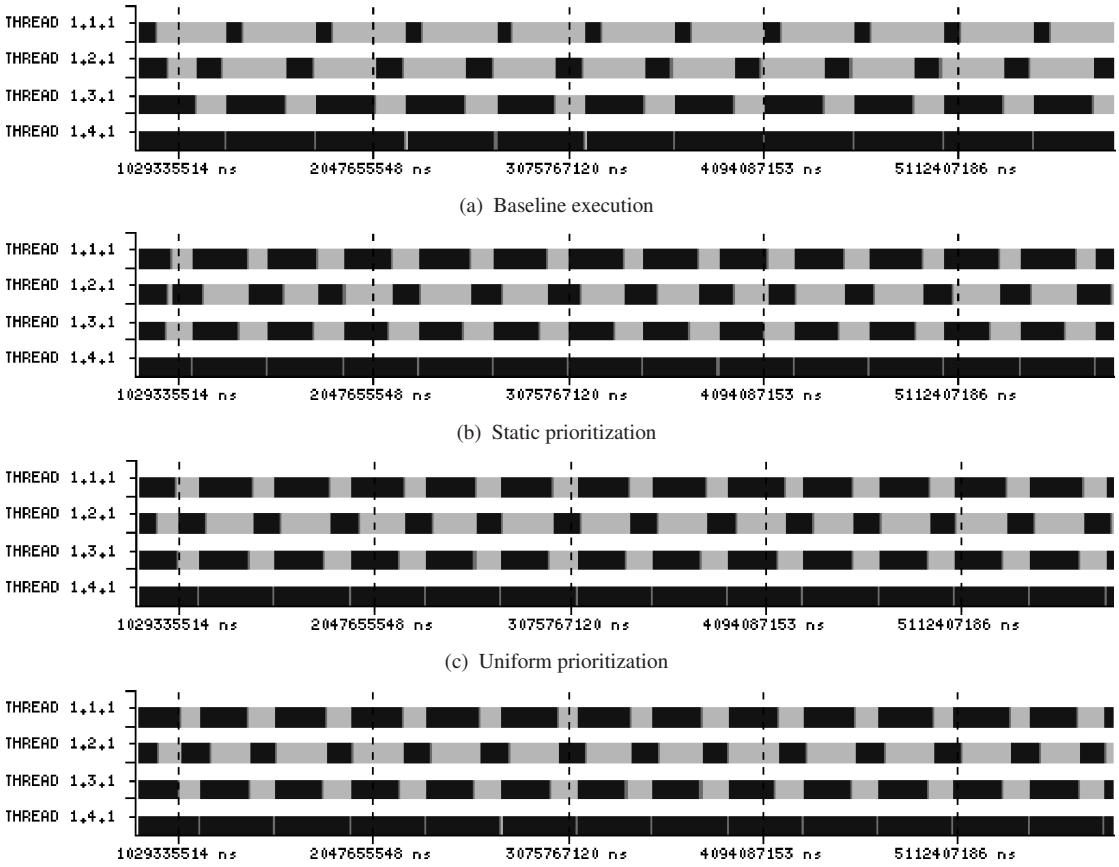
Test	Proc	% Comp	Priority	Exec. Time
Baseline	P1	50.24	4	368.17s
2.6.24	P2	75.09	4	
	P3	50.22	4	
	P4	75.08	4	
Static	P1	99.97	4	338.40s
	P2	68.06	6	
	P3	99.94	4	
	P4	68.04	6	
Uniform	P1	91.47	-	327.17s
	P2	95.55	-	
	P3	91.44	-	
	P4	95.33	-	
Adaptive	P1	89.61	-	326.41s
	P2	93.08	-	
	P3	89.99	-	
	P4	95.15	-	

4.4.3 BT-MZ

Block Tri-diagonal Multi-Zone (BT-MZ) is one of the NAS Parallel Benchmarks (NPB) suite. It is better described in Chapter 3, Section 3.5.2. In our experiments we used BT-MZ with default values: class A with 200 iterations.

TABLE 4.3: BT-MZ balanced and imbalanced characterization

Test	Proc	% Comp	Priority	Exec. Time
Baseline	P1	17.63	4	94.97s
2.6.24	P2	29.85	4	
	P3	66.09	4	
	P4	99.85	4	
Static	P1	70.64	4	79.63s
	P2	42.22	4	
	P3	60.96	5	
	P4	99.85	6	
Uniform	P1	70.31	-	79.81s
	P2	37.18	-	
	P3	65.29	-	
	P4	99.85	-	
Adaptive	P1	70.31	-	79.92
	P2	37.30	-	
	P3	65.30	-	
	P4	99.83	-	



(a) Baseline execution

(b) Static prioritization

(c) Uniform prioritization

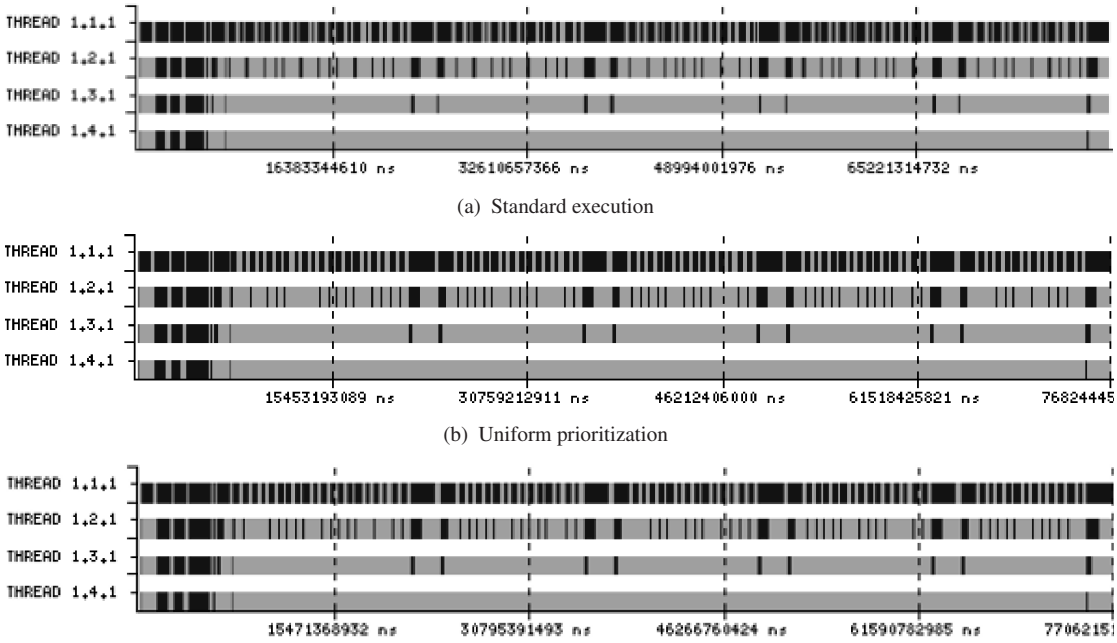
(d) Adaptive prioritization

FIGURE 4.5: Effect of the proposed solution on BT-MZ. Each trace represents only some iterations of the application.

Figure 4.5 shows how HPCSched is able to balance BT-MZ achieving results similar to the static prioritization (Figure 4.5(b)). Both the *Uniform* (Figure 4.5(c)) and the *Adaptive* (Figure 4.5(d)) heuristics are able to balance the application and remain in the stable state. Table 4.3 shows that the performance improvement is about 16% for both heuristics over the standard case shown in Figure 4.5(a)

4.4.4 SIESTA

SIESTA [72] is a method for *ab initio order-N materials simulation*, specifically it is a self-consistent density functional method that uses standard norm-conserving pseudo-potentials and a flexible, numerical linear combination of atomic orbitals basis set, which includes multiple-zeta and polarization orbitals. It is also described in Chapter 3, Section 3.5.3.



(a) Standard execution

(b) Uniform prioritization

(c) Adaptive prioritization

FIGURE 4.6: Effect of the proposed solution on SIESTA.

In this experiment we used the*benzene* particle as input set and we noticed that the application presents an imbalance caused by both the algorithm and the input set (see Figure 4.6(a) and Table 4.4). SIESTA behavior, however, is not constant during each iteration, as can be seen in Figure 4.6(a) and an iteration is not necessarily representative of the next one; this variability decreased the effectiveness of our static balancing.

As can been seen in Table 4.4, both the *Uniform* and the *Adaptive* heuristics are only able to reduce the load imbalance marginally (the CPU utilization of each task slightly increases). However, the HPCSched is able to improve the application's performance, reducing the total execution time from $81.49sec$ to $76.82sec$ for the *Uniform* heuristic and $76.91sec$ for the *Adaptive* heuristic. In both cases the improvement is about 6%.

Clearly this improvement does not come from load imbalance reduction but from the other components of our solution, in this case, from the scheduler policy. Figure 4.6(a) shows that the execution phases are very small and that the tasks need to exchange several messages. While waiting for an incoming message, tasks sleep and need to be waken up as soon as the message arrives. The time between the arrival of the message and the moment the task resumes its execution is called *scheduler latency*: SIESTA is very sensible to this kind of *OS noise*. With the CFS scheduler, whenever a task becomes runnable, it has to compete with all the other processes in the system for the CPU. An `SCHED_HPC` task that wakes up, instead, has to compete only with the other tasks in its class: considering our initial assumption (i.e., usually only one HPC task

per-CPU at any given time) the task is able to immediately run on the CPU, thus, its scheduling latency is reduced.

TABLE 4.4: SIESTA balanced and imbalanced characterization

Test	Proc	% Comp	Priority	Exec. Time
Baseline	P1	98.90	4	81.49s
2.6.24	P2	52.79	4	
	P3	28.45	4	
	P4	19.99	4	
Uniform	P1	98.81	-	76.82s
	P2	53.38	-	
	P3	31.41	-	
	P4	21.68	-	
Adaptive	P1	98.81	-	76.91s
	P2	53.40	-	
	P3	31.47	-	
	P4	21.71	-	

4.5 Conclusions and future work

HPC applications are, in most of the cases, *Single Program Multiple Data* (SPMD), meaning that all processes execute the same code on different data sets. Because of load imbalance these applications do not reach their synchronization points at the same moment, as they are supposed to do.

In [9] we showed how assigning more hardware resources to the most intensive task in an MPI application can reduce the load imbalance and improve performance. We performed this study with a static, hand-tuned approach. In this chapter we proposed a new dynamic solution for balancing HPC application, HPCSched. We implemented our solution as a new task scheduler for Linux 2.6 kernels composed by three components: the scheduling policy (`SCHED_HPC`), the metrics and heuristics (Uniform and Adaptive) and the hardware mechanism.

The heuristic used to balance the tasks in the parallel application is critical to achieve good results: in this chapter we showed that the perfect heuristic depends on the application's characteristics and that constant applications may not react very well with an aggressive, high-responsiveness heuristic and vice-versa.

We tested our new Linux scheduler on an IBM POWER5 machine using four different applications: Metbench, a suit of micro-benchmarks, MetbenchVar (which performs

like Metbench but with different periods of execution), BT-MZ, from the NAS benchmarks suite, and SIESTA, a real application. The results we obtained are good, though they depend on the used heuristic. Our solution works well for constant application like Metbench or BT-MZ providing good results (12% and 16% of performance improvement, respectively). For applications that changes their behavior at run time, HPCSched achieve good performance compared with what a programmer can manually do: MetbenchVar shows a performance improvement of 11% while SIESTA an improvement of about 6%. Our previous static approach we could improve the overall execution time by 8% but that solution required the programmer to manually balance the application while HPCSched is able to balance the application automatically.

We also showed that the improvement comes from a combination of two factors: the scheduling policy and the load balancing.

As future work we plan to expand our solution at cluster level: in fact, HPCSched is a task scheduler able to balance HPC application inside a node but modern Supercomputers consists of Thousands of nodes. In this case there is another level of load balancing which consists of assigning the correct group of tasks to each node (*gang scheduling*) considering that the local scheduler (in our case HPCSched) is able to dynamically assign more or less hardware resource to each task. Moreover, we would like to find an heuristic capable of performing well (even if not optimal) for both constant and dynamic applications.

In the next chapter, we present an alternative solution to HPCSched, DLRB. It provides similar functionality, but is implemented at the user level, with a minimalistic kernel infrastructure.

Chapter 5

A User-Level Load and Resource-Balancer for HPC Applications

5.1 Introduction

In Chapter 3 we showed that the hardware prioritization mechanism of POWER5™ processors can be used to balance HPC applications. First, we ran a 4-tasks MPI[1] application on a POWER5 with equal priorities and detected which processes, on average, computed longer. Then, we manually increased the priority of the longer tasks, increasing their speed and reducing the load imbalance. With this methodology, the prioritization is applied by hand to processes at the beginning of the execution and each process runs with the same priority throughout its execution. An improvement of 8% was obtained on real HPC applications like SIESTA [2]. and 15% for BT-MZ [48], one of the NAS benchmarks.

In a later step, in Chapter 4, we proposed a dynamic solution (HPCSched) implemented as a new task scheduler for Linux 2.6 kernels. With this solution, the OS automatically establishes the hardware priority to be assigned to each HPC process. The technique is transparent to the user: the only modification in the application source code concerns the scheduling policy. For applications that show iterative behavior, HPCSched

[1]Recall that he Message Passing Interface (MPI) [3] standard is the most common programming model used in HPC systems.

achieves similar results to the ones obtained by statically tuning the applications' priorities. Moreover, even if the application cannot have a speed-up, it suffers no slow down.

For HPCSched, the ideal heuristic depends on the application's behavior. We proposed two different heuristics but expected to require more types to cover all kinds of applications. The problem is that a user could hardly implement and install a new kernel heuristic by himself, as the number of available scheduling policies was fixed in the kernel code, and developing new heuristics implied kernel development and debugging. Furthermore, detecting, from kernel level, applications behavior is harder than at the application level. A possible solution to the latter issue would be to implement a system call that indicates to the kernel when iterations are starting or finishing. This would require application's change, and/or a linked periodicity detector that could monitor the application.

In this chapter we present DLRB, an application-level resource balancing mechanism that uses the prioritization interface exported by a minimalistic non-intrusive kernel patch and fits between the application and the MPI interface. DLRB is implemented as a linked library and has several components (see Section 5.2) that can be individually changed, among them, an application periodicity detector. It has some key differences, disadvantages and advantages from the kernel-level solution previously proposed. The main advantages of DLRB, over the previous proposals are:

- As a linked library, each user can choose a different version of the library to suit a specific program. Although several heuristics can be present at the kernel level, there was no easy way to allow the user to implement new heuristics to run with a new application. We found this limitation to be important.

  Strictly speaking, a user-level library is not necessarily easier to deploy or expand, when compared to a kernel scheduler, at least not in a way that can be easily proved or compared. However, we believe that, at least in our environment, there is much more expertise in debugging and implementing user-level code than code in the static kernel sections.

  In addition, installing a new kernel-level extension usually requires an administrator password (for instance, the password for *root*). It is seldom the case that on large shared systems, users have the permissions to change the kernel-level code.

- Finally, if poorly implemented, a badly behaving heuristic can be better constrained to only effect a limited number of users (the ones using this specific

implementation) than a distributed kernel bug, which could eventually crash several nodes.

A disadvantage of a higher-level scheduler is a much bigger granularity. While the kernel scheduler runs every few milliseconds and decides the priority for one thread at time, by design and to avoid performance degradation, DLRB runs at the end of every iteration, which typically lasts for seconds or even minutes. DLRB, however, benefits from a global view of the application, and decides the priorities based on the behavior of all threads.

As in the previous three chapters, we use the POWER5 prioritization system described in Chapter 2 (Section 2.3.2). In addition, to allow the user-level mechanisms to access the full range of hardware priorities, DLRB requires some support from the kernel. It uses the same minimalistic non-intrusive kernel patch described in Section 2.4.3.

The rest of this chapter is structured as follows: Section 5.2 describes DLRB, our user-level load and resource balancer for HPC applications. Section 5.3 shows our experiments on benchmarks and real applications. Finally Section 5.4 provides our conclusions and finalizes the chapter.

5.2 The DLRB

In this chapter we present a user-level Dynamic Load and Resource Balancer (DLRB). DLRB uses the knowledge of the underlying architecture (threads or cores) to perform load-balance, moving tasks across the cores, and resource-balance, changing the distribution of resources between the hardware threads of the same core.

Our solution does not require any change to the application's code and is presented as a a dynamically linked library. It can be linked to the application or loaded at run time with the LD_PRELOAD environment variable present in most UNIX environments. Once linked to the program, DLRB will be triggered by the MPI calls, both from Fortran or C, and will try to detect iterative behaviors in the application. Once an iterative behavior is detected, DLRB will try to redistribute the tasks within the available CPUs in the node and then further decrease the load-imbalance by applying the thread prioritization mechanisms. Furthermore, its structure allows the implementation of several heuristics or optimizations.

In this section, we will better describe each of the DLRB components and its heuristics. Section 5.2.1 describes the Dynamic Periodicity Detector (DPD) that is responsible for

detecting the application's iterations. Section 5.2.2 presents the load balancer, which appropriately places the tasks in the processors. Finally, the resource balancer responsible for the threads prioritization is described in the Section 5.2.3.

5.2.1 DPD

Real HPC applications often present iterative behaviors. To detect this behavior, DLRB implements the Dynamic Periodicity Detector (DPD) proposed in [33].

In their work ([33]), the authors show that, by observing the MPI and OpenMP behavior of an application, DPD is able to correctly identify applications' iterative parallel structures. It presents a very small overhead between 0.012% and 0.064% in most cases, except for hydro2d, one of the SPECfp95 benchmarks, where the overhead was 3.27%.

Having a good detection mechanism for the application's iterations is a key element of DLRB. Detecting the iterations in a precise manner allows DLRB to stabilize the prioritization system on program phases and to properly read the unbalance for individual iterations.

5.2.2 Load-Balancer

The biggest advantages of DLRB are leveraged when it can use one or several cores in a node. In this case, its first step will be to distribute the tasks in the node in a way that balances the sum of the tasks' loads across the cores. This step is performed by the load-balance module of DLRB.

The load-balancer implements an abstraction of domains similar to the one used inside the Linux Kernel. In Figure 5.1, we show the domain organization for a hypothetical machine with two chips, each chip being dual-core and dual-thread. For every logical processor, there is a domain, and for every group of domains, there is another, higher domain, comprehending these domains.

The goal of the load-balancer is to distribute the number of tasks and their load, in terms of utilization, between sub-domains of a same domain, going top-down, from the whole system domain to the chip domains, core domains and, finally, thread-domains.

During the first iteration of the HPC application, the loads are not known for the tasks, and therefore a first blind load-distribution is performed. Knowing only the number of tasks per domain, the load-balancer will try to divide equally the number of tasks to run

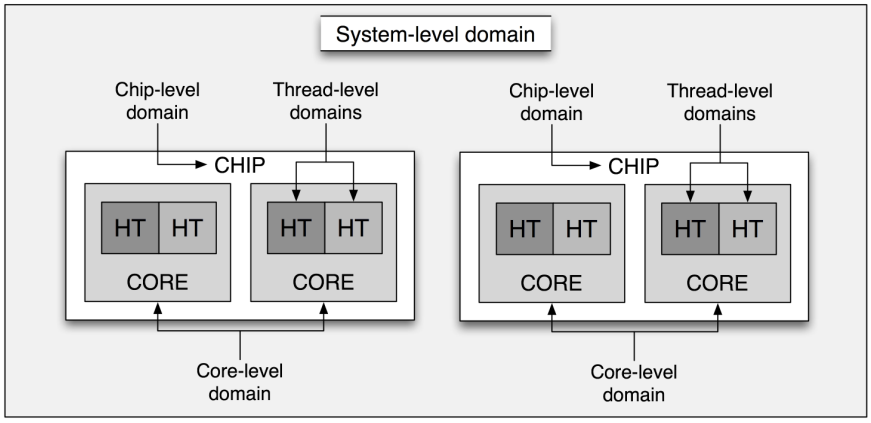


FIGURE 5.1: DLRB scheduling domains for load-balancing.

into the domains of the same level. As for instance, a typical system will execute one MPI task per CPU or hardware thread. After this step, every hardware thread will be running exactly one MPI task.

Before the second iteration, once the utilization of every task is known, the load balancer will migrate the tasks in order to balance the loads in a way that every domain has roughly the same total utilization (summing up all the sub-domains utilizations). This will lead, whenever possible, to a situation where cores will have a high-utilization task in one hardware thread and a low-utilization task in the other hardware thread.

For instance, in a system with one dual-core, dual-thread chip, suppose a program has four tasks (t_1, t_2, t_3, t_4) and they exhibit respectively the percent utilizations (15,23,50,99). At the first iteration, the tasks' utilizations are still unknown. They will be scheduled as follows: t_1 will run on the first thread of the first core, t_2 will run on the first thread of the second core, t_3, on the second thread of the first core and, finally, t_4 will run on the second thread of the second core.

Once the utilizations are known, DLRB will realize that this distribution is not optimal, as the first core has lower utilization than the second core, and that this difference could be alleviated by migrating tasks. Tasks t_1 and t_4 will be coscheduled on the first core, and t_2 and t_3 in the second core. This distribution will persist for the entire program execution, unless a significant behavior change is detected.

Although there is an important architectural overhead from moving one task from one core to another, or even between chips, we understand that usually HPC applications present many iterations and, furthermore, may use a large memory footprint. Therefore, having the risk of migration during the first iteration of a program iterative phase is not a prohibitive impact and the benefits of having a good architectural placement are, by

far, higher. In our experiments, the appropriate placement of the tasks represented a huge impact on the application execution time.

In Chapter 4, we determined that one of the key factors to perform load or resource balance of an application is to determine which are the tasks limiting the performance. For this reason, in that work, the HPCSched only prioritized the tasks with utilization higher than 90%. In DLRB, at every iteration, when the loads of the tasks are evaluated, a bitmap of the tasks with utilization higher than 95% is quickly generated. When an iteration finishes, the bitmap from this iteration is compared with the previous utilization bitmap, if the pattern of the high-utilization tasks is changed, DLRB checks for a behavior change and re-evaluates the load-balancing. As the utilizations of a task is affected by the resource-balancing, the application is run without applying prioritization (resource-balance) during one iteration after a phase change is detected.

While the load-balancer only runs when the program changes its utilization behavior, the resource-balancer is run every time there is unbalance between threads of a core. Its logic is explained in the next section.

5.2.3 Resource-Balancer

Every time an unbalance is detected between the two hardware threads of a core, the resource-balancer tries to use the underlying thread prioritization mechanism to assign more hardware resources to the thread running longer. If there is more than an MPI task running on a hardware thread, the sum of their utilization will be accounted as the hardware thread's utilization.

This mechanism is fairly simple. While the difference of the two thread's utilization is higher than a threshold, the priority of the thread with higher utilization will be increased. If the thread with higher utilization is already at the highest priority, because it already reached the priority configured as maximum, then the priority of the thread with lower utilization will be decreased (unless it reaches the priority defined as minimum). In order to converge faster to the right priority difference, an unbalance compensation is used: if the unbalance is higher than two times the threshold, the higher utilization thread will be prioritized by two priority steps, increasing its priority two times, or decreasing the other thread's priority if needed. In this work, we used priorities ranging from four to six, inclusively, and an unbalance threshold of 20%[2].

[2]In Chapter 2 we show that priority differences higher than two may incur in significant performance degradation for the low priority thread.

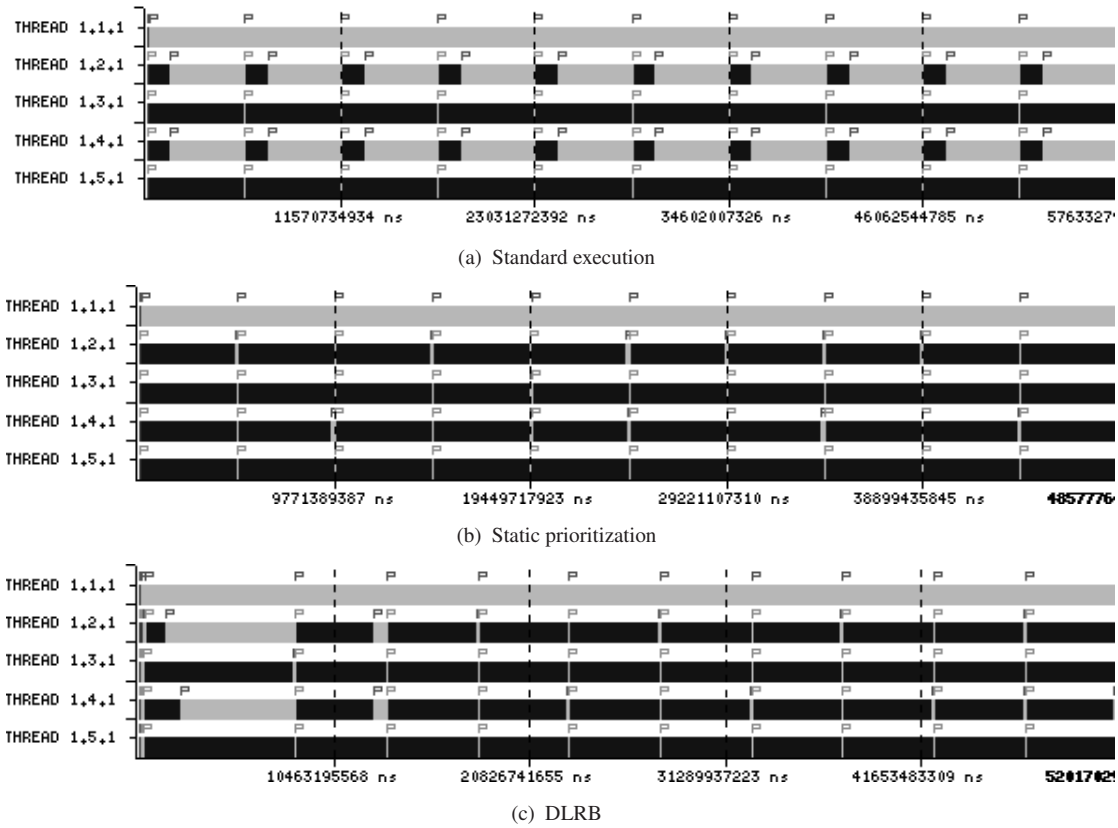
5.3 Experiments

In this section we evaluate the performance of DLRB, our user-level load and resource balancer, and compare it to the standard environment, where no prioritization is used, the static solution proposed in Chapter 3 and the best result obtained, in each situation, from the two heuristics of HPCSched proposed in Chapter 4

We present results for Metbench, our micro-benchmark suite (Section 5.3.1), MetbenchVar (Section 5.3.2), BT-MZ (Classes A, B, C) from the NAS benchmark suite (Section 5.3.3) and SIESTA (Section 5.3.4).

The experimental infrastructure used is similar to the previous two chapters. The key difference is that we used updated versions of MPI (MPI-CH 1.0.8) and Metbench/MetbenchVar (1.1a). As a performance metric, we evaluate the total execution time.

5.3.1 Metbench



(a) Standard execution

(b) Static prioritization

(c) DLRB

FIGURE 5.2: Effect of the proposed solution on Metbench.

In this section we present the results, when running Metbench as previously described in Chapter 3, Section 3.5.1.

Figure 5.2(a) shows part of the execution trace of our reference case, where Metbench runs with the standard Linux scheduler. Recall that, in this figure dark areas represent the computing time, while gray areas represent the waiting or communication time. In fact, two of the Metbench workers are idle for about 75% of the time. Figure 5.2(b) shows the solution proposed in Chapter 3, where we were able to statically balance the application: the execution time is improvement by about 15%. The static approach we used in [9] require previous knowledge of the application and effort from the programmer to detect the load imbalance and to properly assign hardware resources to each task.

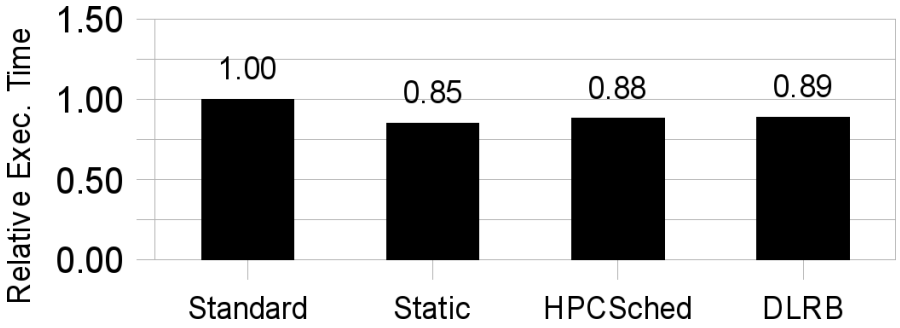


FIGURE 5.3: Relative execution times for ten iterations of Metbench.

Figure 5.2(c) shows the execution of DLRB. It is able to converge to the appropriate prioritization in two iterations, detecting the stable state of the application and maintaining the best allocation until the end of the the program execution.

Figure 5.3 shows the relative performance of the standard execution (100%), the static prioritization (85%), HPCSched (88%) and DLRB (89%).

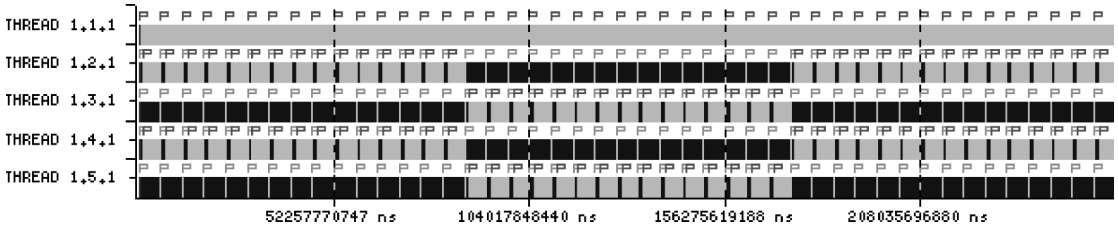
Two issues are very important here:

1. the DLRB converges slightly slower than HPCSched, as it needs to wait until the end of every iteration to detect the unbalance. Furthermore, it runs the first iteration of a phase without applying any prioritization to test the load distribution. This extra iteration represents a penalty to the achieved improvement of the solutions based on DLRB. In fact, this effect is severely alleviated if Metbench is ran for a larger number of iterations, as for instance, with a hundred iterations, the static prioritization achieves 85% of improvement and DLRB achieves 85.5%.

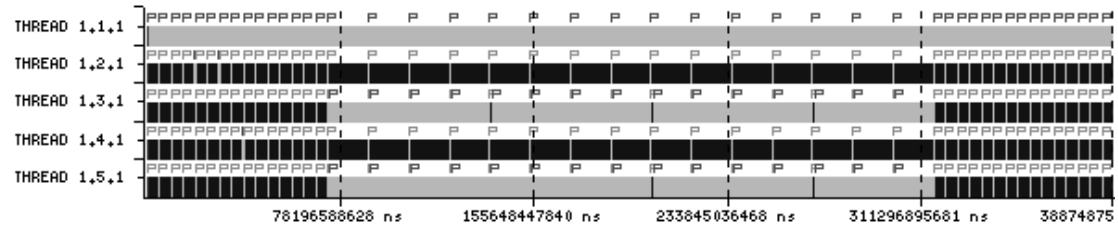
2. For a relatively small number of iterations, the initial placement of the tasks matters and the speed of the first iteration is influenced by how the tasks are coscheduled. The example shown is the worst case, where the two high utilization tasks are scheduled in the same core until DLRB obtains their relative utilization and redistributes the loads.

Observe that the first iterations on Figure 5.2(c) are slightly longer than the first iteration on Figure 5.2(a). Metbench has five tasks (t_1 to t_5), the first one, the master, has utilization close to zero and waits for the workers are performing their duty. The following four tasks are the workers, which, in this case have utilizations (25,99,25,99). At the load-distribution step the utilizations are unknown and t_1 and t_5 are placed on hardware thread one, t_3 is placed on thread two, t_2 on thread 3 and t_4 on thread four. Unfortunately, tasks two and four reveal to be the tasks with higher utilizations. Before the second iteration the utilizations are known and the tasks are redistributed, t_4 runs on the first thread, t_3 and t_0 on the second, and t_2 on the third and t_1 on the last thread.

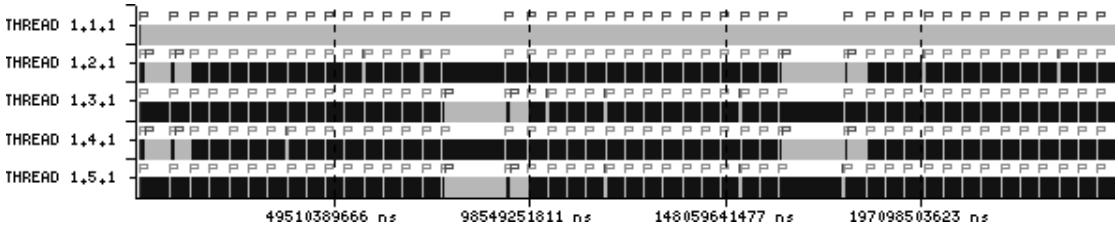
5.3.2 MetbenchVar



(a) Standard execution



(b) Static prioritization



(c) DLRB

FIGURE 5.4: Effect of the proposed solution on MetbenchVar. Observe that the static prioritization has a different time scale.

MetbenchVar is described in Chapter 4, Section 4.4.2. As we mentioned before, it is a modified version of Metbench (in this case, Metbench 1.1) that allows the workers to change their behavior after a configurable number of iterations. Figure 5.4(a) shows the default execution of this benchmark, when no prioritization is applied.

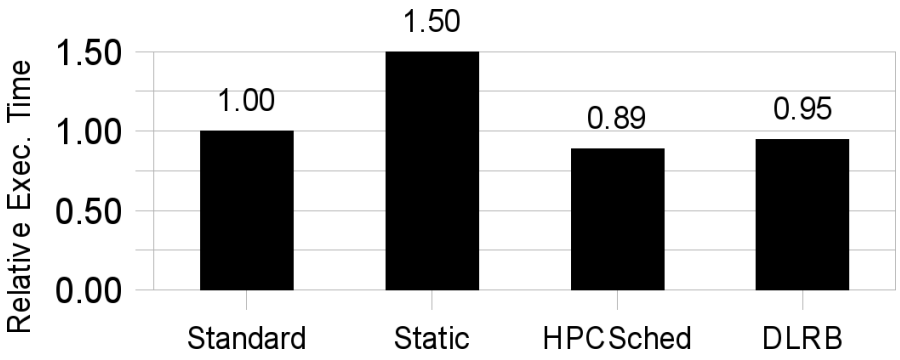


FIGURE 5.5: Relative execution times for 45 iterations of MetbenchVar (changing behavior every 15).

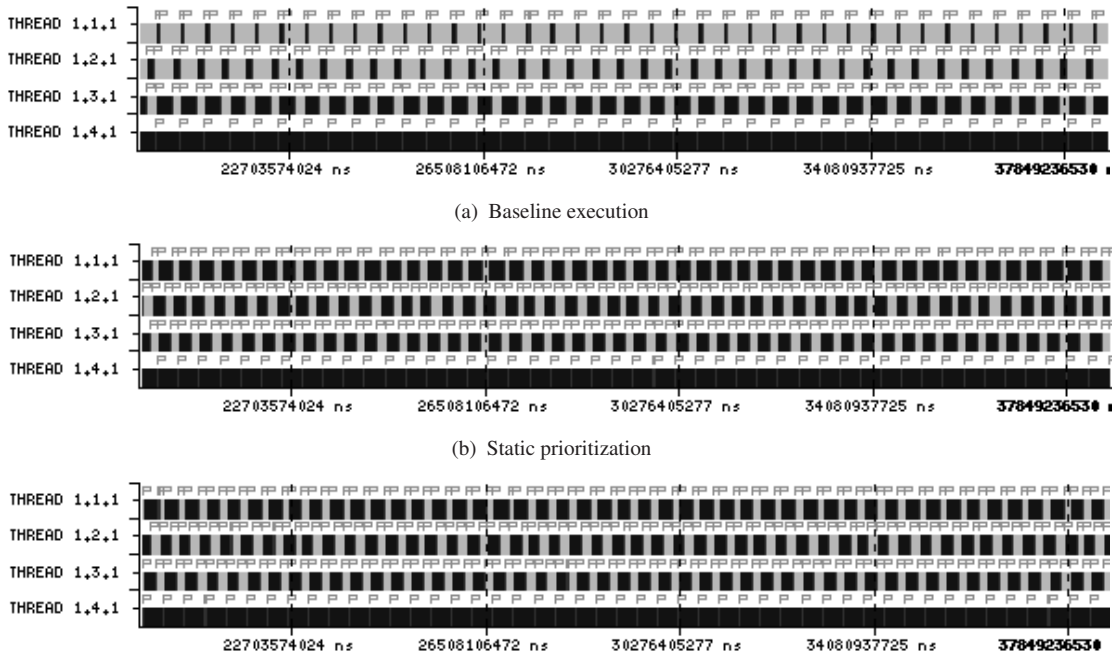
Figure 5.4(b) shows that, for this workload, the negative impact of applying the wrong prioritization is extremely high and, although for $\frac{2}{3}$ of the cases the benchmark runs with the right priorities (4,6), the performance degradation of running with the wrong priorities is by far more important. Figure 5.5 shows that overall, the static prioritization presents a 50% performance degradation when compared to the standard case of this benchmark. HPCSched obtains 11% of improvement.

Having an application that responds very negatively to wrong prioritization is the worst case scenario, when a task has drastic changes in the utilization phases. Because DLRB needs to wait until the end of a iteration to calculate the unbalances, it executes an entire iteration before detecting that the application behavior changed (see Figure 5.4(c)). Once it detects a change, it redistributes the tasks across the processors, then the application runs for another iteration without applying the priorities (see Section 5.2.2) and finally assumes the correct priorities for the next iteration (this is a two iteration convergence time). Because of this overhead, DLRB only obtains 5% of improvement for the benchmark. When MetbenchVar runs with phases of 45 iterations, for 180 iterations in total, DLRB obtains a performance improvement of 12%, comparing to the standard execution time. In both cases, DLRB converges to the right priority in two iterations and is able to maintain a stable state during the rest of the application phase.

5.3.3 BT-MZ

Block Tri-diagonal Multi-Zone (BT-MZ) is one of the NAS Parallel Benchmarks (NPB) suite. It is better described in Chapter 3, Section 3.5.2.

Figure 5.6 shows the execution of the standard execution, the static prioritization, DLRB. Because of the granularity of the prioritization mechanism, DLRB is not able to fully balance the application. Giving priorities four and six is not enough to fully balance the



(a) Baseline execution

(b) Static prioritization

(c) DLRB

FIGURE 5.6: Effect of the proposed solution on BT-MZ. Each trace represents only some iterations of the application.

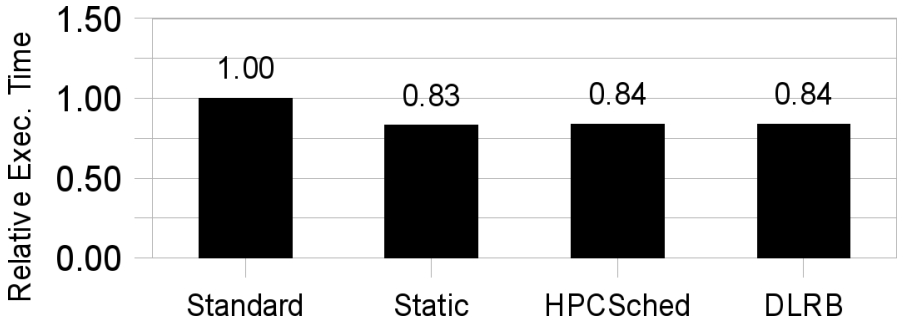


FIGURE 5.7: Relative execution times for BT-MZ class A.

loads, whereas priorities six and three would over-prioritize the task with high utilization and represent a significant performance loss (see Chapter 3).

As we can see in Figure 5.7, the static prioritization yields 17% of performance improvement, while HPCSched and DLRB both yield 16%. For classes B and C, the static prioritization shows respectively 16% and 17% of improvement, while the rest of the solutions show 16% of improvement.

5.3.4 SIESTA

SIESTA [72] is described in Chapter 3, Section 3.5.3. It is a real application ran at the BSC. Its behavior depends on the analyzed material and there is a plethora of available inputs.

For our experiments, we analyze the *benzene* particle (the same presented in the previous chapter). With this material, the application presents a significant unbalance but only a small region of the application presents iterative behavior. For most of SIESTA's execution, there is not an iteration that is representative of the behavior of the next one. The solution presented in the previous chapter was not effective to balance this application, but represented a performance improvement of 6% (See Figure 5.8). This improvement came from the fact that the HPCSched increased the responsiveness of the application and decreased the OS noise.

As shown in the Figure 5.8, DLRB obtains 5% of improvement. A result similar to the one obtained by HPCSched.

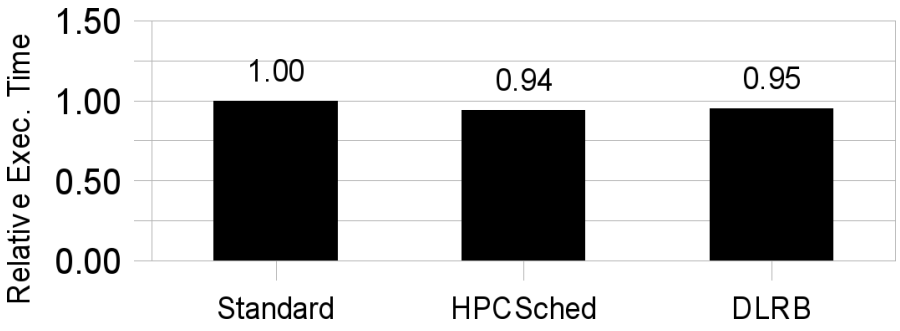


FIGURE 5.8: Relative execution times for SIESTA.

5.4 Conclusions and future work

In the previous chapters, we presented the use of resource balancing, through hardware-prioritization, as a way to decrease the unbalance presented by HPC applications (Chapter 3) and developed a task scheduler for the Linux 2.6 kernels that, using one of the two proposed heuristics was able to reduce the load imbalance and improve applications' performance (Chapter 4). We concluded that the ideal heuristic depends on the application.

Although it is possible to add new heuristics in the kernel, it is hard to allow the users to deploy new kernel-level heuristics on large clusters. Furthermore, debugging and development at the kernel level is harder than at user-level.

In this chapter, we provided a user-level load and resource balancer, DLRB. It only requires a slim infrastructure at the kernel level, to allow the hardware priorities to be accessible from the user-level and provides an infrastructure that makes possible to the user to implement new heuristics or scheduling policies that best fit their specific needs. The mechanism proposed reaches similar results than the best heuristic of HPCSched in most cases and does not present performance degradation.We conclude that the current proposal is able to schedule both constant and dynamic applications with similar results as our kernel scheduler, while presenting a higher level of flexibility.

As a future work, we would like to improve other metrics, like energy or temperature reduction and using other hardware mechanisms to perform the resource balance.

Until this point of this thesis, we approached the problem of coordinating the hardware resource allocation with the software targets for the high performance computing domain. In the next chapter, we analyze the problem of scheduling tasks in a soft real-time SMT system. As there are no commercially available hardware with explicit resource allocation for soft real-time, we developed and used a simulator that allowed to perform the system's task scheduling, taking into account the resource sharing in the underlying simulated architecture.

Chapter 6

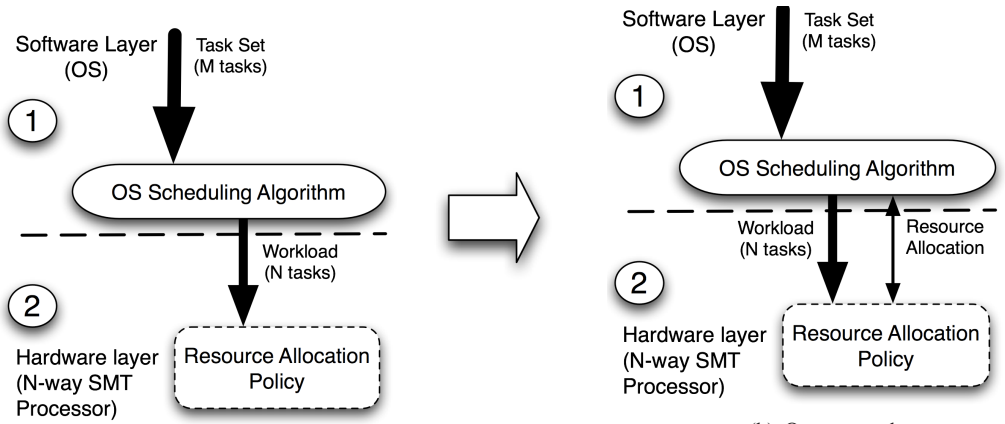
Scheduling for Soft Real-Time SMT Systems

6.1 Introduction

In this last chapter, we approach the problem of scheduling tasks in soft real-time systems on SMT processors.

SMTs architectures have demonstrated to provide high-performance at a relative low cost [68][76] and have motivated their use in high-performance processors [49][61]. SMT processors adapt a superscalar front-end to fetch from several threads while the back-end is shared. They have high throughput but poor performance predictability.

The scheduling of a task set in such processors involves two main steps as shown in Figure 6.1(a). In a first step, known as *workload selection* [47], the Operating System (OS) scheduler selects a set of N tasks from the task set of M tasks, where N is the number of contexts of the SMT processor and M is usually greater or equal than N. This set of N tasks is called the $workload$. Next, the OS passes the workload to the architecture. In a second step, known as *resource sharing*[47], the SMT internal resource allocation mechanism determines how resources are distributed among threads, and how the threads are prioritized at a hardware level. In current processors this resource allocation mechanism is limited to the instruction fetch policy, like icount [76], DCache Warn [14], *data gating* [32], FLUSH [75] or FLUSH++ [13], while the first step is performed roughly every time slice (typically between 1 and 100ms), the second step occurs every cycle.



(a) Current approach

(b) Our approach

FIGURE 6.1: Collaboration between the OS job scheduler and the SMT hardware: steps required to schedule a task set in classical SMT processors.

The key issue in the interaction between OS and a traditional SMT system is that the OS only assembles a workload of N tasks while it is the processor that decides how to execute this workload, implicitly by means of its internal resource allocation policy. Hence, there are two different schedulers working, without any collaboration with each other, and part of the traditional responsibility of the OS "disappears" into the processor, sometimes reverting software priorities or simply disregarding them. Consequently, the OS may not be able to guarantee time constraints on the execution of a thread if that thread is running concurrently with other threads, even though the processor has sufficient resources to do so. In order to deal with this variability, several hardware (resource sharing policies) and software approaches have been proposed [47] [20] [16] [17] [31] [30] [57] [25].

The objective of this chapter is to consistently bind the OS and the software targets and priorities to the hardware resource allocation in a way that makes resource-sharing become a viable option to increase performance predictability of real-time systems, at a low cost.

In this work we address the problem of scheduling a task set in a SMT system from the software and hardware layers in a collaborative way. Our proposal allows better control of the underlying hardware resources (like the issue queues or the registers) by the scheduling algorithm, increasing the task scheduling success rate. Assuming that the Worst Case Execution Time (WCET) is given, for every task, our mechanism does not require any additional profiling.

The original *E*arliest *D*eadline *F*irst (EDF) [56] and *L*east *L*axity *F*irst (LLF) [26] algorithms only aim to determine the order in which threads should be executed. This is not

enough if the task set is scheduled on a SMT processor due to the execution time variability of threads. We developed and evaluated two new scheduling algorithms, called SRA-EDF (Single-objective Resource-Aware EDF) and SRA-LLF (Single-objective Resource-Aware LLF), that use the hardware support proposed in [16]. These resource-aware algorithms, in addition to determine the execution order of threads, determine the amount of resources given to co-scheduled threads. They are provided with the knowledge of the processor resources and instruct it in how to split resources among threads in order to meet the system deadlines. This increases the success rate when scheduling tasks, outperforming state-of-the-art scheduling algorithms. In addition, we propose a new hardware mechanism that allows to optimize a second objective function. When running with the new hardware support, the schedulers were called DRA-EDF and DRA-LLF (*D*ouble-objective *R*esource-*A*ware EDF and *D*ouble-objective *R*esource-*A*ware LLF).

This work is structured as follows: Section 6.2 presents some background on real-time scheduling and the related work; Section 6.3 explains our proposal; Section 6.4 presents our methodology and experimental setup, while Section 6.5 provides the experimental results; finally Section 6.6 is devoted to the conclusions.

6.2 Background and Related Work

This chapter focus on real-time SMT scheduling for independent tasks. In this case, real-time systems are characterized by a group of tasks, called a *task set*. For each task, the scheduler knows three main parameters: first, the *period* (p_i), that is, the interval at which new instances of a task are ready for execution. Second, the *deadline* (d_i), that is, the time before which an instance of the task must complete. For simplicity, the deadline is often set equal to the period resulting into an implicit-deadline system [37] [38]. This means that a task has to be executed before the next instance of the same task. Third, the *Worst Case Execution Time* ($WCET_i$) is an upper bound of the time required to execute any instance of the task, which should never be exceeded (for single threaded executions). In the scope of this work, the WCET is known a priori and is not considered profiling.

In soft-real time scheduling, many algorithms (e.g., EDF [56] or LLF [26]) have been used to schedule a task set in single-threaded systems. However, these algorithms are no longer sufficient on SMT processors, since the execution time of a thread is unpredictable when this thread is scheduled with other threads. The high variability of SMTs

implies that a real-time job scheduler for SMT processors is much more complex and challenging than for single-threaded processors.

Another problem is that algorithms like EDF are not optimum when scheduling on a multiprocessor system, scheduling anomalies like the Graham anomaly [39] or the Dhall effect [27] may occur. In our perception, a SMT system is a variable heterogeneous multiprocessor and therefore presents these anomalies, even on a single chip: the fact that one single SMT processor or core presents multiple execution flows makes it equivalent, from the scheduler perspective, to a multiprocessor. In addition, each execution flow has a variable performance, depending on the characteristics of the tasks coscheduled in the different threads.

6.2.1 Workload composition

In [47] the authors make a detailed design space exploration of scheduling algorithms. From the many proposed algorithms, the best one is GLOB_SYM_US, while a second algorithm, called GLOB_NOSYM_US presents the best relation between performance and complexity. GLOB_SYM_US is actually a hybrid implementation that defaults to GLOB_NOSYM_US when at least one task has $U_i > \frac{N}{2N-1}$, being U_i the utilization of a task τ_i and N the number of available hardware contexts, otherwise, it defaults to GLOB_SYM_PLAIN. The latter extends EDF selecting first the task with earliest deadline, and then, for the other N-1 tasks, assigns the tasks in order to maximize the symbiosis factor of the running task set, which is defined in [70] as:

$$symbiosis\ factor = \sum_{i=1}^{N} \frac{\text{realized IPC of } \tau_i}{\text{single-threaded IPC of } \tau_i} \tag{6.1}$$

Here, the number of Instructions per Cycle (IPC) is used as a measure of performance. Hence, the higher is the symbiosis factor, the better should be the processor pipeline utilization and, therefore, the higher the gain of throughput due to the SMT. This algorithm is tuned to give the best processor utilization. Note, however, that this requires the profiling of every N-way task combination (from the task set of M tasks) in order to find task sets with best symbiosis, which leads to a number of profiles equal to $C_r(M,N) = \frac{(M+N-1)!}{N!(M-1)!}$. For instance, with our 10 benchmarks, we needed to profile 55 different combinations. If, instead of 10, we had 50 different benchmarks, we would need to profile 1275 different combinations. Besides, if we take into consideration that the IPC of a workload may change depending on the offset of the running threads, the profiling complexity increases by orders of magnitude. Running all the combinations

of the MediaBench benchmarks (as shown in the Table 6.2), in our hardware configuration, yielded a maximum symbiosis of 1.70, a minimum of 0.94 and an average of 1.40.

GLOB_NOSYM_US (also called EDF_US [73]) extends EDF by giving higher priority to tasks with utilization greater than $\frac{N}{2N-1}$ (deadlines are set to $-\infty$ and ties are broken arbitrarily).

In [57], the authors make an interesting theoretical study of scheduling algorithms for SMT processors. They propose a scheduling mechanism for both non-adjustable and adjustable processors. The authors define adjustable processors as processors that allow some kind of resource allocation, they give as example the one proposed in [18]. Their algorithm activate and deactivate the hardware contexts, going sometimes from a single-threaded configuration to an 8-way SMT processor in order to adjust the relative speed of threads. In this study it is considered that the different tasks simultaneously running on the processor contexts share equally the resources, which is not always the case as shown in [18]. In contrast to this study where no hardware model is simulated, we provide performance results from a cycle-by-cycle simulator (*smtsim* [76]). This allows us to accurately take into account the inter-task interferences.

6.2.2 Resource allocation

Several hardware mechanisms have been proposed in order to bias the execution of a thread in a given workload with different degrees of success. In [71] an extension to the *icount* fetch policy is proposed by including handicap numbers that reflect the priorities of jobs. Although this mechanism is able to prioritize threads to some extent, execution times of jobs are still hard to predict, making this approach unsuited for real-time constraints.

In [47], the authors focus on workload selection in soft-real time systems, although they also briefly discuss the resource sharing problem. The authors use two types of resource sharing: dynamic and static. As a dynamic resource sharing they use the icount fetch policy. In the static resource sharing the authors statically profile the performance of each job with only the allocated resources in single-threaded mode. They assume that the IPC of a job only depends on the resources allocated to it and not on the co-scheduled jobs. This information is passed to the scheduling algorithm to find a feasible schedule. The authors conclude that the dynamic resource sharing achieves better success rate than the static one but at the cost of schedulability.

Dorai [28] propose transparent threads, which is a mechanism that allows background threads to use resources that a foreground thread does not require for running at almost full speed. Their main metric is the multithreading level, which evaluates how many threads are active for a given scheduling through some period of time.

In [6], the authors propose an approach suitable for hard real-time where the WCET is specified assuming a virtual simple architecture (VISA). At execution time, a task is executed on the actual processor. Intermediate virtual deadlines are established based on the VISA. If, during its execution, a task fails to meet its intermediate deadlines, the processor is reconfigured to implement the VISA, bounding the execution time of the task. If the actual processor is an SMT and a task fails to meet its intermediate deadlines, the SMT is switched to single-threaded mode, to ensure that tasks can meet their deadlines. The authors conclude that fetch policies that attempt to maximize throughput, like *icount*, should be "balanced" to guarantee minimum forward progress of real-time tasks. This is precisely the target of our work: we ensure a minimum amount of resources for a given time-critical thread so that it meets its deadline regardless of the other threads executed in its workload. Our approach is orthogonal to the VISA framework: a time-critical thread is executed on the actual SMT processor that provides the thread with a given percentage of resources.

Using VISA, in the event that a task does not meet its intermediate deadlines, the processor switches to single-threaded mode. Here, we do not have intermediate deadlines, but tasks get an amount of resources that should guarantee that they meet their deadlines. This amount is recalculated and is readjusted every time the scheduler runs. On unlikely cases, it may even give all the resources to an urgent task, running the processor in single-thread mode.

In [16], several mechanisms were proposed to allow the software to establish the amount of resources to give to the critical thread, controlling in this way the interaction among threads, and the slowdown suffered by each thread in SMT mode. The difference among these mechanisms is the information required from the application: the higher the information used by the hardware mechanism, the better the results, and the more complex the mechanism is. However, in that paper no scheduling algorithm was proposed. That is, the responsibility of determining the amount of resources to give to each thread so that it meets its deadline is left to the OS.

To our knowledge, there is no work aimed to bind the OS prioritization to the hardware priorities in such a coordinted way. Either resource aware hardware or software schedulers were proposed. In this work, we aim to extend a software scheduler in order to

show that making this bridge is not only possible, but also profitable and desirable. We developed a simulation environment that allows us to use a larger number of software threads than available hardware contexts, we evaluated costs of context switches and implemented different system schedulers, binding the task priorities to the hardware allocation.

Such scheduler can be implemented in any explicit resource aware processor, or even on SMT processors featuring priority control, like, for instance, IBM POWER5™ [49]. However, for this research we chose to use a simulated environment as, to our knowledge, there is currently no commercially available processor with explicit prioritization for real-time systems.

6.3 Our Proposals

In this chapter we propose two scheduling policies that take profit of SMT in-processor resource-allocation mechanisms to guarantee better schedulability. As a baseline, our new scheduling algorithms, SRA-EDF and SRA-LLF, use the hardware support proposed in [16], called *LVP* or *Low-Variability Performance*. In addition, we propose a new hardware support that allows the optimization of two targets: a minimum resource allocation and the maximization of a second function, as for instance, throughput.

SRA-EDF and SRA-LLF allow a closer collaboration between the software level and the SMT hardware. This tight collaboration shows many advantages: First, it achieves better success rate than all the proposals previously explained in the Section 6.2. Second, no additional profiling, other than the WCET estimation, is required from applications to carry out the scheduling task, assuming that there is an estimate of the Worst Case Execution Time (WCET) of the tasks. Furthermore, when shared resources cannot be controlled by software, it is often the case that the internal hardware prioritization mechanism goes on the opposite direction of the OS priorities, for instance, giving fetch priority of the task with lesser OS priority. Our mechanism fully avoids this situation, as it binds the OS priorities to the hardware mechanism. We start by explaining in detail the underlying hardware support used as a baseline, the LVP mechanism.

The basis of the hardware mechanism proposed in [16] is to partition the hardware resources between the threads running on a SMT and reserve a minimum fraction of the resources for a designated *Most Critical Thread* (MCT), enabling it to meet its deadline. In that work, the authors proposed two hardware resource allocators denominated *static* and *dynamic* LVP (Low Variation Performance). These allocators differ on what

information the hardware mechanism expects to receive from the OS [1]. In the *static* approach, it is assumed that the OS task scheduler provides a resource allocation that is fixed for an entire period. While in the *dynamic* approach, it provides the target IPC for the MCT. In the latter approach the resource allocator can dynamically vary the amount of resources dedicated to the critical thread. The Predictable Performance (PP) hardware, proposed in [17] differs from the *dynamic* LVP version as it receives the percentage of the performance the thread must be run. That is, it takes into account different program phases, being able to dynamically scale both resource allocation and the thread IPC.

We chose to implement our algorithms with the *static* LVP version of the hardware. We do not implement the *dynamic* LVP or the PP hardware approach because, although they provide better results, they are more complex and have lower applicability than the *static* LVP.

6.3.1 Overall functioning

When the WCET of a task is determined, it is assumed that this task has full access to all the underlying platform resources. However, when this task runs with other tasks in a multithreaded environment, it only uses a certain fraction of the resources. When the amount of resources given to a thread is reduced, its performance may decrease as well. The relation between the amount of resources allocated to a program and the performance is different for each program and may vary for different inputs of the same program. In [16] it was observed that in a SMT system the relation between the amount of resources given to a thread and its *relative performance* [2] follows a "super-linear" relation. That is, if we reserve X% of resources to a given thread its relative performance is greater than or equal to X% of its performance when having all the resources. It was also observed that the main shared resources to take into account are the physical registers, the fetch bandwidth, and the instruction window. The proposed hardware splits the shared hardware resources among running threads as indicated by the OS task scheduler. That is, it allows the OS to specify the amount of resources to use by each thread.

Our scheduling algorithms, SRA-EDF and SRA-LLF, use the hardware support proposed in [16] to take profit of this relation. When the OS level task scheduler wants

[1] Recall that [20],[16] and [17] only focus on the hardware part and do not deal with the workload composition problem.

[2] The relative performance is the IPC that a thread has when it is given X% of SMT resources, with respect to its performance when it is run with all the resources. It ranges between 0 and 1.

to execute a critical task τ_i, given its $WCETst_i$ (Worst Case Execution Time in Single Thread mode) and a deadline d_i, it simply computes the allowable performance slowdown that, initially, is represented by, $S_i(0) = \frac{WCETst_i}{d_i}$, and instructs the hardware to reserve, for that hardware context, a percentage of the resources equivalent to $S_i(0)$[3]. For such a value of $S_i(0)$, each instance of this task should finish before its deadline, supposing that the real execution time of this instance is its $WCETst_i$. Hence, $\frac{T_i}{S_i(0)} = \frac{d_i}{WCETst_i} \cdot WCETst_i = d_i$. Refer to Section 6.3.2 to further analysis on the allowable slowdown calculation.

The proposed methods use global scheduling: tasks are not bound to contexts of a SMT and can be executed in any of the available contexts. For each workload we found the MCT, the thread with the highest priority according to the scheduling algorithm under study. The MCT is evaluated every time the running workload changes (whenever there is a context-switch on a context). Therefore, at any given moment, there will be exactly one thread running as MCT (also said that this thread has the MCT status) and as we use a 2-context SMT, there will be another thread as LPT.

DRA-EDF and DRA-LLF operate in a very similar manner. They still specify the allowable slowdown S_i for the MCT, but due to the additional hardware support, the processor is free to give it more resources, tuning the resource allocation to yield the highest performance at a secondary metric. In this work we chose the processors aggregated throughput as a secondary metric.

6.3.2 The SRA-EDF scheduler

This algorithm improves the normal EDF scheduler [56] in order to make it resource aware. It adds the concept of a most critical thread (MCT) and a lower priority thread (LPT) running together in the SMT and it is aware of the sharing of hardware resources across the processor contexts.

SRA-EDF starts filling contexts, putting first the task with the closest deadline and, therefore, the highest priority. This task is considered to be running as the most critical thread (MCT). Then, the second closest deadline will occupy the second hardware context, being the second highest priority task or, to keep the notation of [16], the LPT. The MCT will receive an amount of resources large enough to guarantee its deadline,

[3] $S_i(t)$ stands for the allowed slowdown that a thread can have to still fulfill its deadline at a given instant t

while the LPT will receive the rest of the available resources. This logic can be easily expanded to a n-way multithreaded processor as long as the first thread receives an amount larger or equal to the resources needed to reach its deadline, while the other threads can be considered as LPT and share the remaining resources.

After some time, one of the following things may occur:

- The task with nearest deadline finishes its execution. In this case, the second closest deadline becomes the next deadline and, therefore, the previous LPT becomes now the MCT. The resource allocation to the MCT context is re-evaluated and the highest priority task receives the amount of resources necessary to fulfill its deadline. The next task with the closest deadline is put in the newly free context.
- The LPT finishes before the MCT. The next task with the closest deadline is put in the newly free context.

As we can see, during its execution time, a task can run as MCT, LPT or, more likely, both (note that the status are mutually exclusive at a given instant). The resource allocation that the MCT receives at a given period of time ($RA(t)$) is calculated on the allowed slowdown ($S_i(t)$) that a task τ_i can take at an instant t, in order to fulfill its deadline, while the LPT runs with the remaining processor resources: $RA(0) = S_i(0)$. The allowed slowdown is evaluated every time the running workload changes or when the scheduler runs (typically at time slices boundaries). In other words, whenever tasks are changed on any of the hardware context, the $S_i(t)$ for the thread with the nearest deadline (MCT) is evaluated or adjusted.

Conceptually the $S_i(t)$ calculation is very simple as it consists on the ratio between the Remaining Computation time for a task τ_i (RC_i) and the remaining time to its deadline (TTD_i), i.e:

$$S_i(t) \; = \; \frac{RC_i}{TTD_i} \tag{6.2}$$

However, there are some considerations to be made on each of its factors, as we will see below.

The Time to Deadline (TTD) is always evaluated as the difference between the deadline d_i of a task τ_i and the current time t ($TTD_i = d_i - t$). However, we must take into account its range. When a task crosses its deadline boundary, the TTD_i becomes negative, invalidating the resource allocation calculation. The appropriated action to be taken in this case may vary according to the system. If it makes no sense to continue, one may want to kill tasks that missed their deadlines, this is the case when processing

video frames. On the other hand, for other applications, it may be interesting to give them full priority, when the deadline is somehow malleable. Finally, it may also be desirable to give them the minimum priority, understanding that the task is probably close to finish or the last evaluated priority (probably very high).

The Remaining Computation time (RC_i) evaluation is related to the difference between the the total amount of work to be done and the one that was already done. The simplest way to represent it, given our available data would be: $RC_i = WCETst_i - running\,time$, where $WCETst_i$ represents the Worst Case Execution time, in single-thread, for a given task. The running time can be evaluated in different ways, according to how we take into consideration the impact of the resource sharing on the performance of the MCT and the LPT.

The first option, is to consider them equally and simply use the number of cycles a task has been scheduled in a hardware thread. In this case, the S_i calculation would be expressed as follows:

$$S_i \;=\; \frac{WCET_i - \sum_{\gamma=1}^{l}(\omega_\gamma)}{d_i - t} \tag{6.3}$$

Where $\sum_{\gamma=1}^{l}(\omega_\gamma)$ represents the sum of all intervals of size ω_γ that a task τ_i ran in a hardware thread.

We found that better results could be achieved improving this calculation as we must consider that the processor was not entirely dedicated to only one task, and therefore the processing already done for a task should consider the resources allocated to the task.

One step further is to consider the resource allocation a task received while running. Assuming the relation between the resource allocation and the performance of a task is linear, we simply calculated the processing done as the product of the time the task was running and the resource allocation given to it. Using this concept, we obtain the following formula:

$$S_i(t) = \frac{WCET_i - \left(\sum_{\gamma=1}^{m}(\omega_\gamma * RA_\gamma) + \sum_{\gamma=1}^{l}(\omega_\gamma * (1 - RA_\gamma))\right)}{d_i - t} \tag{6.4}$$

Here, $\sum_{\gamma=1}^{m}(\omega_\gamma * RA_\gamma)$ represents all the resources that a given task τ_i received as a MCT, that is, the sum of all the intervals of size ω_γ ran with resource allocation RA_γ when τ_i was the top priority task of the system. In addition, we have $\sum_{\gamma=1}^{l}(\omega_\gamma * (1 - RA_\gamma)$ as the total amount of processing done when running as LPT.

In [16], it was shown that the performance for the MCT is super-linear[4]. As a consequence, the relation among resources given to the LPT and its performance may, sometimes, be sub-linear. Based on this conclusion, we chose to evaluate the remaining computational time for a given task τ_i with different functions whether the task is running as a MCT or LPT. When running as LPT, the worst case is assumed and the performance is considered sub-linear. This correction basically makes the time a thread runs as low priority accounts for less processing time than while it is running as a MCT. We believe that this is the most accurate evaluation. To summarize, the allowed slowdown for a task τ_i, is given by the following formula:

$$S_i(t) = \frac{WCET_i - \left(\sum_{\gamma=1}^{m}(\omega_\gamma * RA_\gamma) + \sum_{\gamma=1}^{l}(\omega_\gamma * (1 - RA_\gamma^f))\right)}{d_i - t} \tag{6.5}$$

Formula 6.5 differs from Formula 6.4 on the calculation of the devoted resources for the LPT as they are now evaluated as $\sum_{\gamma=1}^{l}(\omega_\gamma * (1 - RA_\gamma^f)$. f is an empirical constant aimed to reduce the total accounted resources for the LPT.

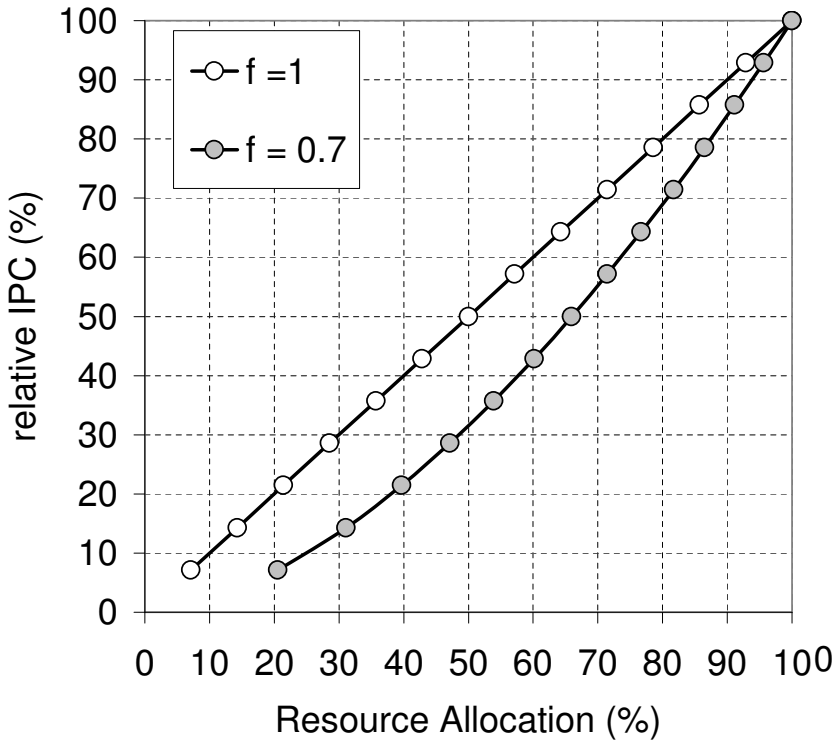


FIGURE 6.2: Accounted performance for LPT based MCT RA.

[4]Recall that the MCT receives priority over the LPT when fetching instructions from the instruction cache.

We found, based on the results of [16] and on empirical data, that 0.7 is the most appropriated value for the constant f. In Figure 6.2 we can see the accounted IPC for LPT when the relation between the LPT performance and resources allocated to the MCT is linear ($f = 1$) or sub-linear ($f = 0.7$). As we can see, the same amount of resources for both $f = 1$ and $f = 0.7$ translates into a lower performance accounted for the period a task was executed in LPT mode..

The results, when using Formulas 6.3 and 6.4, show much less flexibility and the success rates were significantly smaller. In all cases, the formula 6.5 yielded better results. For simplicity, we only present experimental results for this evaluation method.

Furthermore, we observed that, for task sets with many tasks of relatively distant deadlines, the allowed slowdown can be very significant, allocating less than half of the resources to the MCT. In such cases, we understand that there is no sense on giving less than 50% of the resources to the thread that is, by definition, the highest priority of the workload. We artificially constrain the Minimal Resource Allocation ($MinRA$) as 0.5. Hence, $0.5 \leq RA(t) \leq 1.0$ for the RA version of our proposed algorithm. The MCT resource allocation at a given moment is expressed as follows:

$$RA(t) = \max(MinRa, S_i(t)) \tag{6.6}$$

It is also very important to observe that, even if the $S_i(t)$ calculation may seem complex, it is done in software, by the OS task scheduler and the above described sums are simply implemented as accumulators that are only updated when the resource allocation for the MCT changes (because of a context-switch, for instance) and the entire value of $S_i(t)$ is only evaluated for the MCT.

6.3.2.1 Example

Consider a set of 4 tasks with the deadlines, WCETs, and utilization as shown in the Table 6.1. In a first step, the scheduler chooses the task τ_1, the task with closest deadline. It becomes the MCT. The operating system (OS) also chooses to run, at lower priority, τ_3, which is the task with second closest deadline. The OS finally assigns a RA to the MCT. It is set to the $MAX(S_1, 0.5)$ and S_1 is calculated as $S_1 = \frac{1-0}{5-0} = 0.2$. Therefore, the MCT receives $RA = 0.5$.

Assume that, at a given instant, say $\Upsilon = 1.25$, τ_1 finishes. τ_3 becomes the task with the closest deadline and the MCT. The context where τ_1 was running, receives now τ_2.

TABLE 6.1: Hypothetical task set.

Task	WCET	Deadline	Utilization
τ_1	1.0	5	0.20
τ_3	4.0	6	0.67
τ_2	0.5	7	0.07
τ_4	2.0	8	0.25

S_3 is calculated as $\frac{4-(0+(1.25*(1-0.5^{0.7}))}{6-1.25} = 0.74$, $RA = MAX(0.74, 0.5) = 0.74$. This workload executes until the instant $\Upsilon = 3.8$, when τ_2 finishes. τ_4 starts to run on the free context and the new RA for τ_3 is calculated as follows: $RA = MAX(S_3, 0.5)$ where $S_3 = \frac{4-((2.55*0.76)+(1-0.5^{0.7}))}{6-3.8} = 0.76$. This behavior repeats during the entire execution time.

6.3.3 The SRA-LLF scheduler

Our SRA-LLF (Least laxity first) algorithm follows the same logic as the SRA-EDF. For each given workload, the S_i is calculated in the same way and follows the same constraints. However, recall that LLF periodically calculates the laxity of each task, being the laxity: $l_i = (TTD_i) - (RC_i)$. Recall that TTD_i is the time to the deadline ($deadline_i - actual\,time$) of task τ_i and RC_i is the remaining computation time necessary to complete this task.

As discussed in the previous section, the RC_i can be computed in several ways. We chose the RC_i calculation method used to determine the S_i, explained in Formula6.5 (Section 6.3.2) and applied it to the laxity evaluation. Therefore, following the terminology of the section 6.3.2 the final laxity formula was:

$$l_i = TTD_i - \left(WCET_i - \left(\sum_{\gamma=1}^{m} (\omega_\gamma * RA_\gamma) + \sum_{\gamma=1}^{l} (\omega_\gamma * (1 - RA_\gamma^f)) \right) \right)$$

Again, the sums are the same accumulators that already existed for the S_i calculation and represent no additional complexity for the algorithm.

Concerning task preemption, taking executing tasks out of the run queue while running a EDF based scheduler did not make sense, as our deadlines are fixed at the beginning of the execution, and, therefore, the priorities on the EDF would never change. For the LLF, however, it is possible that a running task looses priority while running, if its laxity becomes larger than another task that was previously evaluated as having more slack

time. Similar to the EDF based versions, for which there is no task preemption (tasks execute until they finish as they priority cannot change), we found that our best SRA-LLF implementation did not interrupt running tasks when they do not have the least laxity any more. Although the switch time (including scheduling) is often neglected [37], our results shown that schedulers with higher switch rates yielded to worse overall results.

A small correction was, however, found to be very useful to avoid the need to preempt a running task. At any workload change (when one of the tasks finished), a new task was inserted (being the task with the highest priority among the ready tasks) and the two running tasks had their laxities compared between them to determine which of the two would assume the role of MCT. In this way, the highest priority task in the processor is set to the task-set highest priority task.

6.3.4 A hardware support for improved scheduling

In LVPs static approach, the MCT is allowed to use a given amount of resources established by the OS, which is called *Minimum Resource Allocation* or *MRA*. The other thread was allowed to use the remaining $(1 - MRA)\%$ of resources. If any thread use more resources than given to it, that thread was stalled until it frees resources.

A characteristic of this hardware mechanism is that it was *single-objective* in the sense that the hardware could only satisfy a single target function given by the OS. In this case the objective was to ensure a minimum resource allocation to the MCT.

Our novel resource allocation, works with the assumption that the slowdown factor S_i is the minimal bound for the slowdown that the MCT can suffer. Therefore, we can provide the MCT with any amount of resources x so that $x \geq S_i$. This is acceptable to guarantee the deadline of the task τ_i.

Enabling the resource allocator to use any value $x \geq S_i$ provides a new degree of freedom to the hardware, allowing it to maximize a second objective function given by the OS. For this reason, we call our resource allocator DRA that stands for Double-Objective Resource Allocator.

In our approach, the OS provides the workload, a minimum resource allocation to achieve for the MCT and a second objective function. In this work we use as a second objective function the IPC throughput or sum of IPC of co-scheduled workloads.

Once the time-slice starts, our mechanism splits the execution of the given workload into two intervals or states that are executed in alternate fashion as shown in Figure 6.3.

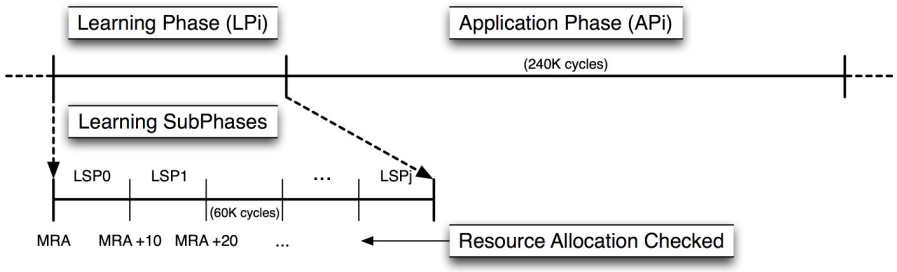


FIGURE 6.3: States and sub-states in which our resource allocation divides the execution of a workload

During the first state, the *Learning State* the resource allocator tries different resource allocations greater than or equal to the minimum resource allocation provided by the OS for the MCT. The resource allocation leading to the highest value of the second target function is chosen and used during the application state and maintained until the next learning state. In Figure 6.3, we see an example where RA_i is set to MRA and the increment per step in the resource allocation (RA_{inc}) is set to 10%, which is the value used in our experiments.

The learning state is split in sub-states. In the first learning sub-state the MCT is given $RA_i + (0 \times RA_{inc})$ resources. In the second sub-state the MCT is given $RA_i + (1 \times RA_{inc})$, and so on while $RA_i + (k \times RA_{inc}) \leq 100\%$. The number of learning sub-phases or steps until 100% of the resources are allocated to the MCT depends on the minimum resource allocation (RA_i) set by the OS and the size of the increment for each step (RA_{inc}).

The duration of an application period is set 30 times larger than a learning sub-state and the learning period is set to a maximum of 6 sub-states, as for the DRA versions the lower bound for the resource allocation on the most critical thread is 0.4 ($0.4 + (6 * 0.1) == 1$).

We coupled RA-EDF and RA-LLF to this new hardware support. The resulting combination of scheduling algorithms and hardware support were respectively called DRA-EDF and DRA-LLF. Between a SRA and a DRA version of a resource-aware algorithm, the only difference lays in the underlying hardware mechanisms and implementation.

6.3.4.1 Hardware Implementation

The DRA based schedulers require additional hardware support to track the phase and sub-phase in which the mechanism is in. This hardware is similar to the mechanism

proposed in [23], and can be done with a hardware-scheduler thread (or *hs-thread*) that is programmed in micro-code.

APP_DURATION:	OS established value to indicate number of cycles of the application phase.
LEAR_DURATION:	OS established value to indicate number of cycles of the learning phase.
RA_{inc}:	OS established value to indicate the value incremented to the RA_{curr} at each learning sub-state.
RA_i:	OS established initial value of RA.
RA_{curr}:	register that establishes the resources to give to the MCT.
counter:	countdown counter: decremented of 1 at each cycle.
inst:	counter that increments at each committed instruction.
app_period:	Boolean variable that indicates whether it is in application or learning phase.
best_inst:	storage for the best result on an application sub-state.
RA_{best}:	storage for the resource allocation that obtained the best result.
RA_{avgi}:	storage for the internal accumulator for the average resource allocation from the last RA set to now.
RA_{avge}:	storage for the average resource allocator (visible to the OS) of the last period between two RA sets.

FIGURE 6.4: Registers and variables needed by the hs-thread.

The pseudo code for the hs-thread can be seen in the Figure 6.5, while the variables needed by the algorithm are described in the Figure 6.4. The first piece of code (Figure 6.5(a)) is executed every time a new resource allocation (RA_i) is defined by the OS and passed to the processor, usually when switching tasks. It consists of few simple instructions where, only in the line 6, the assigned value is not an internal value. The first branch tests in which phase the processor was and, according to it, accumulate on the internal register RA_{avgi} the amount of resources given on the last running period. The line 6 sets the internal RA_{curr} to the RA_i, which is the value passed from the OS. The 7th makes the internal counter receive the value of the learning duration, the 8th line makes the hardware enter in learning period and the last one updated the visible special register that indicates the total average resource allocation for the last period between two RA sets. This value may be used by the OS to calculate the next allowed slowdown. Otherwise, the OS wouldn't be able to determine the exact amount of resources that each thread received on the last time-slice.

Whenever *counter* reaches zero, the running threads (MCT and LPT) are stopped and, the second part of the code (Figure 6.5(b)) is executed in order to reset the amount of resources given to each thread. If it is the end of the application period or the end of a learning-sub period, different actions can be taken. In the first case, it executes the code block from line 2 to 5: it accumulates the internal average resource allocation given to the MCT on the last application period, sets the counter to the size (in cycles) of a learning sub-state and re-set the hardware to the learning period, it also sets the RA_{curr}

```
1 if(app_period){
2   set RA_avgi to RA_avgi + (RA_curr × (APP_DURATION - counter))
3 }else{
4   set RA_avgi to RA_avgi + (RA_curr × (LEAR_DURATION - counter))
5 }
6 set RA_curr to RA_i;
7 set counter to LEAR_DURATION;
8 set app_period to false;
9 move RA_avgi to RA_avge;
```

(a) Processing when the resource allocation is re-set.

```
 1 if(app_period){ /* end of app phase */
 2   set RA_avgi to RA_avgi + (RA_curr × APP_DURATION)
 3   set counter to LEAR_DURATION;
 4   set app_period to false;
 5   set RA_curr to RA_i;
 6 }else{ /* end of a learning period */
 7   if(inst > best_inst){
 8     set best_inst to inst;
 9     set RA_best to RA_curr;
10   }
11   set RA_avgi to RA_avgi + (RA_curr × LEAR_DURATION)
12   add RA_inc to RA_curr;
13   if(RA_curr > 1.0){
14     set app_period to true;
15     set RA_curr to RA_best;
16     set counter to APP_DURATION
17   }else{
18     set counter to LEAR_DURATION
19   }
20 }
```

(b) Processing done at the end of each phase (invoked when *counter* reaches zero).

FIGURE 6.5: Hs-thread pseudo-code.

back to the to the RA_i(line 5). On the other hand, if it is the end of a learning sub-period, the hardware stores the best resource allocation until the moment (lines 8,9), accumulate the resources given to the MCT on the last learning sub-period (line 11), and increment the RA_{curr} by RA_{inc} (line 12). This procedure repeats, on every learning sub-state boundary, until the RA_{curr} gets bigger than 1.0, when it sets the *app_period* to *true* (line 12), the RA_{curr} to the RA_{best} and the counter to *APP_DURATION*.

We modeled the internal overhead of this code (in the pipeline) as 50 cycles, however, as one can see, the worst case is composed by 3 branches, 5 assignments, two adds and one multiplication which probably executes much faster. Moreover, this code will only execute at each sub-phase boundary, defined by the OS.

Every cycle the counter is decremented until it reaches zero, at the beginning, *APP_PERIOD* is set as false and, therefore, a learning sub-state will start with a given resource allocation (each time incremented by 0.1). This will undergo until the resource allocation reaches a value superior to 1.0. At that time, the *APP_PERIOD* gets the number of cycles it must execute (we used 240k cycles).

6.4 Methodology and Experimental Environment

In this section we describe the experimental methodology used to evaluate the performance of the proposed and the previous scheduling algorithms. Our experimental setup is similar to the experimental setup shown in [30]. This section covers the definition of the task set, metrics, and the architecture simulator.

6.4.1 Task sets and Metrics

In this work, we use the MediaBench benchmark suite [54]. We compose task sets with 3 different sizes: 4, 8 and 12 tasks randomly chosen from the MediaBench benchmarks shown in Table 6.2. We believe the choice of these task sizes is reasonable. Since in our experiments we use a two-way SMT, a 2-task scheduling defaults to no scheduling needed (as they don't need to be multiplexed between the two hardware contexts) and significantly larger task sets (say of hundreds of tasks), would take too long (weeks) to simulate on a cycle accurate OS/architecture simulator. Right now, one 12-task simulation takes around 6 to 10 hours to execute alone.

TABLE 6.2: MediaBench benchmarks used in this work.

Benchmark name	Media	Language	WCET for a 1GHz proc.	input
adpcm_c	speech	C	1.6772 ms	clinton.pcm
adpcm_d	speech	C	1.4599 ms	clinton.pcm
epic_c	image	C	17.8306 ms	test_image.pgm
epic_d	image	C	6.1524 ms	test_image.pgm
gsm_c	speech	C	55.9323 ms	clinton.pcm
gsm_d	speech	C	50.8701 ms	clinton.pcm
g721_c	speech	C	39.7142 ms	clinton.pcm
g721_d	speech	C	18.1077 ms	clinton.pcm
mpeg2_c	video	C	34.9833 ms	test2.mpeg
mpeg2_d	video	C	2.5358 ms	test2.mpeg

For a given task τ_i the *utilization* is defined as $U_i = WCETst_i/p_i$, where $WCETst$ is the Worst Case Execution Time (WCET) of the task in single-thread mode and p_i is the period of the task. As shown in [47], for a task set the *serial utilization (SU)* is defined as the sum of the utilization of each of its tasks. In other words, given a task set with M tasks:

$$SU = \sum_{i=1}^{M} \frac{WCETst_i}{p_i} \tag{6.7}$$

The term *scalar utilization* is also used in [30] with the exact same meaning. In this work, we will use the term serial utilization.

We evaluate the performance of each scheduling algorithm under different scenarios of increasing difficulty. We vary the serial utilization from 1.0 to 2.6 with a step of 0.2, for a total of 9 scenarios. We do not present the results for serial utilization higher than 2.6 as, even if they were simulated, they fail to add any new information: the processor is already saturated with a 2.6 serial utilization. For each task set size and serial utilization (SU) we created 50 task sets. Thus, for each scheduling algorithm we ran 1350 simulations (3 task set sizes, times 50 task sets, times 9 serial utilization). As evaluation metric we use the *Success Rate*, which measures how many task sets are successfully scheduled. We consider that a task set is successfully scheduled when all tasks in that task set finish before their deadline.

6.4.2 Simulator

We use a trace driven SMT simulator derived from *smtsim* [77]. The simulator consists of our own trace driven front-end and an improved version of smtsim's back-end. It allows executing wrong path instructions by using a separate basic block dictionary that contains all static instructions.

Our baseline instruction fetch policy is *icount* [76]. Instructions are decoded and renamed to track data dependencies. When an instruction is renamed, it is allocated an entry in the window or issue queues (integer, floating point and load/store) until all its operands are ready. Each instruction also allocates one Re-Order Buffer (ROB) entry and a physical register in the register file. ROB entries are assigned in program order and instructions wait in this buffer until all earlier instructions are resolved. When an instruction has all its operands ready, it reads its source operands, executes, writes its results, and finally commits.

Processor Configuration	
Pipeline depth	12 stages
Fetch/Issue/Commit	8 entries
Queues Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	120 integer, 120 fp
(shared)ROB size	512 entries
Branch Predictor	16K entries gshare
Branch Target Buffer	256-entry, 4-way assoc.
Return Address Stack	256 entries
Memory Configuration	
Icache, Dcache	64 KB, 2-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	512 KB, 8-way, 8-bank, 64-byte lines, 20 cyc.acc.
Main memory latency	300 cycles
TLB miss penalty	160 cycles

FIGURE 6.6: Processor and memory configuration for the simulation infrastructure

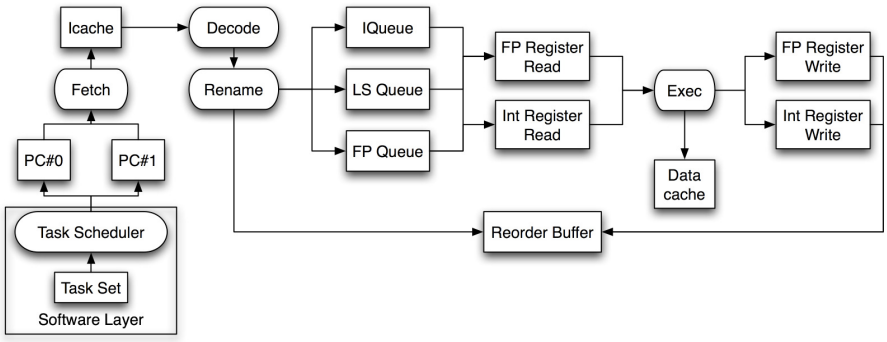


FIGURE 6.7: Schematic view of the simulation infra-structure

We use an aggressive configuration, shown in Figure 6.4.2: many shared resources (issue queues register, functional units, etc.), out-of-order execution, wide superscalar, and a deep pipeline for high clock frequency. These features cause the performance of the processor to be very unstable, depending on the mix of threads. Thus, we evaluate our proposals on an unfavorable scenario. If those proposals work in this hard configuration, they will work better in narrower processors with fewer shared resources. Figure 6.4.2 gives a schematic view of our processor while Figure 6.4.2 shows our baseline configuration.

To be able to validate the system scheduling, we adapted the simulator, allowing it to

receive an input of μ traces ($\mu > \eta$) and multiplex them over the η processor contexts in a way similar to the operating system (OS) task-scheduler. The context-switches are commanded by the task scheduling algorithm and can be timely dependent (say, every 10 or 20ms) or after the execution of a task instance (on a period), according to the scheduler characteristics.

Every context-switch clears the pipeline of the affected context, flushing the active instructions. We also chose to be conservative concerning the memory impact of this switch and assumed the worst case concerning the memory footprint of the task running on the physical context. Therefore, we flush the cache and completely invalidate TLB entries for a context after a context switch, as it is done in some real processors [61]. The evaluation of the Worst Case Execution Time (WCET) in single-thread mode takes into account this overhead. Another key reason for clearing the cache is that the traces may have equal physical addresses, because they were not generated at the same time. In that case, extra care must be taken in order to avoid false hits on the cache after multiplexing some successive traces.

6.5 Experimental Results

In this section we present the results obtained with the several scheduling algorithms implemented and compared in this chapter. Figures 6.8, 6.9 and 6.10 show the number of successfully scheduled task sets for the different scheduling algorithms. We present the aggregated results for the three task set sizes shown in Section 6.4.1, in total, for every serial utilization, 150 different sets were ran. The higher the number of task sets scheduled by an algorithm, the better the result.

As EDF is one of the best known and most common scheduling algorithms for soft-real time systems, we will use it as the main baseline to compare the results obtained by the different algorithms. Recall that SRA-EDF is our EDF based algorithm that runs with the single-objective hardware proposed in [16], while the DRA-EDF refers to the case where the equivalent resource-aware EDF based algorithm runs with our novel double-objective hardware.

The second baseline will be the LLF algorithm, used when comparing LLF the based schedulers. Recall that, SRA-LLF is our LLF based algorithm that runs on the single-objective hardware, and DRA-LLF runs with the double-objective hardware support.

In this work, the results for GLOB_SYM_US and GLOB_NO_SYM_US differ from the ones presented in [47]. This is expected, up to some level, due to several differences in

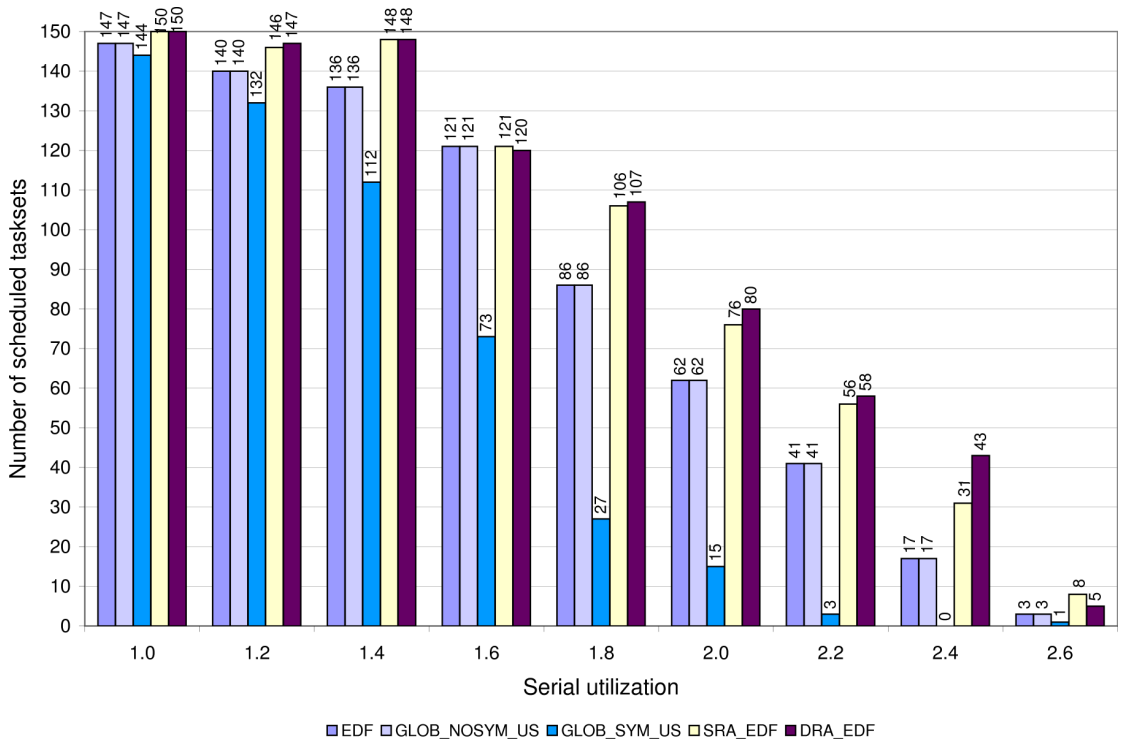


FIGURE 6.8: Comparison between various EDF based algorithms (4-,8-,12-tasks).

the simulated scenario. First, as described in Section 6.4.1, we have different task sets distribution and generation mechanisms. Second, we considered a task set not schedulable when at least one task miss its deadlines, while in [47] the authors considered a task set not schedulable when more than 5% of the tasks have missed deadlines.

Figure 6.8 shows a comparison between the various algorithms based on EDF. The first thing to note is the unexpected behavior of GLOB_SYM_US and GLOB_NOSYM_US. For the latter, recall that its behavior only differs from the EDF for tasks presenting utilization superior to $\frac{N}{2N-1}$. As the percentage of these tasks in our task sets is only of 11.3%, the results of this algorithm are very similar to those of the EDF. In fact, for our infrastructure, it obtains the same results as EDF.

GLOB_SYM_US starts by co-scheduling the task with the earliest deadline (τ_{edf}) and the task that provides the highest symbiosis with this task (τ_{symb}). In [47], authors do not explain how the algorithm performs the rearrangement of the running workload once the τ_{edf} finishes.

It is not clear how the algorithm performed the reconfiguration of the running workload once the task with the closest deadline finishes. In other words, given that the task with the earliest deadline is running on one of the hardware contexts, and the others occupied with the tasks that maximize the symbiosis, many decisions can be taken once the task

with highest priority finishes. We tuned the following factors in order to have the best performance:

- As the task left running is the τ_{symb}, should the algorithm choose the next job in order to maximize the running workload symbiosis or should it pick the task with the nearest deadline?
- Given that, during the previous step, the job with the closest deadline was inserted. If the context running the other (less priority) task did not offer the best symbiosis, should it be preempted in order to have always the best running symbiosis?

We implemented and benchmarked every combination of the previous decisions and concluded that the best option presented the following behavior: Once the task with the closest deadline finishes (τ_{edf}), the one with the next closest deadline is chosen. If this job is the one already running (previously chosen to improve symbiosis), we insert the task that maximizes the symbiosis of the running task set. If this is not the case (the thread with the earliest deadline is not in the SMT processor), then we insert the job with the closest deadline and, if and only if necessary, preempt the other running job in order to run the one with higher symbiosis on the task set. Although this option yields a larger number of context-switches, it proved to be the most efficient version of this algorithm. Still, in the overall, it schedules 32.7% less task sets than EDF.

For EDF, one may observe that some task sets miss even when $SU = 1.0$. This occurs because of a hardware-software priority inversion problem, where the hardware is prioritizing threads in the opposite direction of the software-priorities. The first two cases occurred when *adpcm_c* was running with *epic_d*. As we can see in Table 6.3 (task set 1), the default *icount* policy prioritizes *epic_d*, in order to increase the overall processor throughput, disregarding the fact that *adpcm_c* had a closer deadline (due to the lack of collaboration between the OS and the processor schedulers). For the third case, shown in the Table 6.4 (task set 2) the same problem occurs: *epic_d* was the task with the closest deadline and was scheduled with *epic_c*, of lesser priority. Internally, *icount* policy prioritized the latter in order to increase the overall throughput. Observe that the *symbiosis* factor for these cases are larger than one, meaning that scheduling those tasks together gives a higher throughput than in single thread.

GLOB_SYM_US tends to privilege cases with higher symbiosis, but is not able to tune the internal processor's resource allocation to prioritize the task with closest deadline. It misses three more task sets than EDF and GLOB_NOSYM_US when running with serial utilization 1.0.

TABLE 6.3: Benchmark interactions without explicit resource allocation.

Task set 1:

Benchmark	IPC Alone	IPC Together	Relative IPC
adpcm_c	4.181	1.281	0.306
epic_d	1.642	1.333	0.812

Symbiosis factor: 1.118

TABLE 6.4: Benchmark interactions without explicit resource allocation.

Task set 2:

Benchmark	IPC Alone	IPC Together	Relative IPC
epic_d	1.642	0.787	0.479
epic_c	3.175	1.912	0.602

Symbiosis factor: 1.081

Another interesting fact is that, even if the 4-, 8- and 12-task simulations follow the same trend of behavior, the 12-task sets yield better success rates. This can be explained by the fact that the as the number of tasks per task set increases, the individual per-task utilization utilization tends to decrease, becoming easier to accommodate a larger individual slowdown (traded for a global throughput increase).

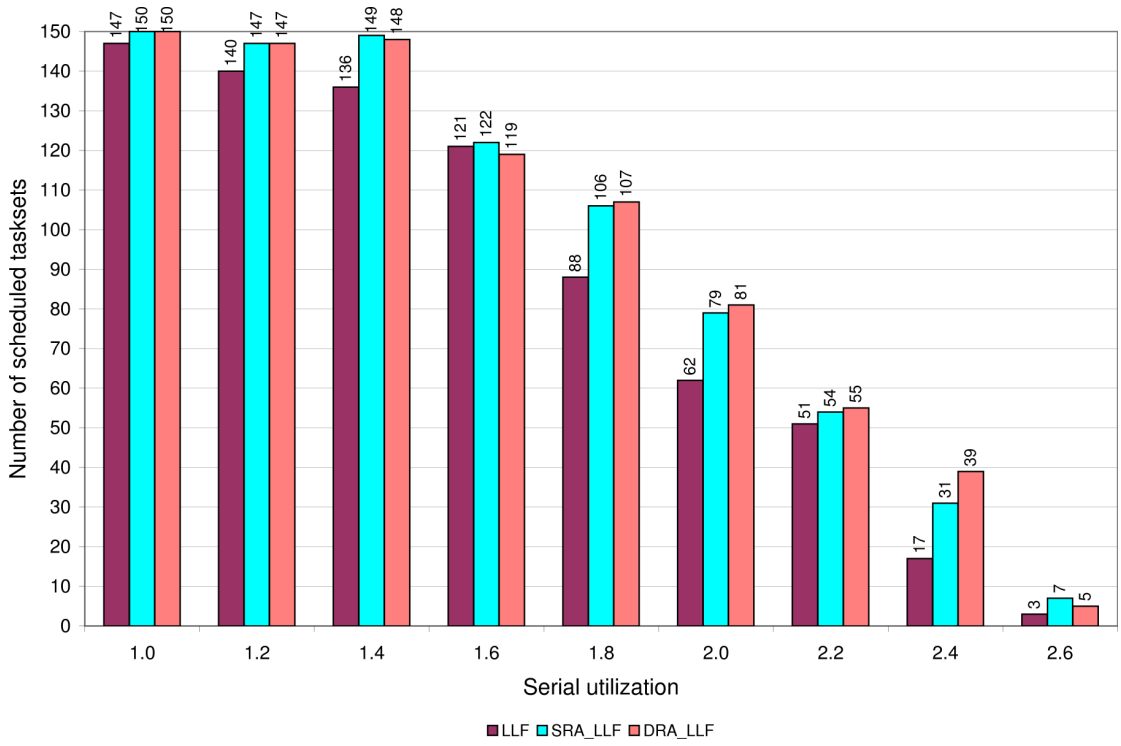


FIGURE 6.9: Comparison between the LLF based algorithms (4-,8-,12-tasks).

As we can see in the Figure 6.8, our RA-EDF algorithm outperforms EDF in all cases. Comparing to the original EDF algorithm, the RA-EDF has, in average, 11.8% higher

success rate[5]. In addition, our SRA-EDF and DRA-EDF successfully schedule task sets when others fail because of hardware-software priority inversion, as they are able to explicitly control the hardware resource allocation to the task with the highest priority. Compared to EDF, DRA-EDF schedules 13.9% more tasks. It achieves similar or higher results than SRA-EDF in all cases except for serial utilizations 1.6 and 2.6, where it schedules respectively one less and three less task sets.

Figure 6.9 shows the LLF based algorithms. When compared to LLF, SRA-LLF schedules 10.5% more task sets, while DRA-LLF schedules 11.2% more task sets. For serial utilizations inferior to 1.8, DRA-LLF seems to be slightly worse than, or equal to, SRA-LLF. From 1.8 to 2.4, it is better than SRA-LLF. LLF schedules about 1.6% more task sets than EDF (Figure 6.11).

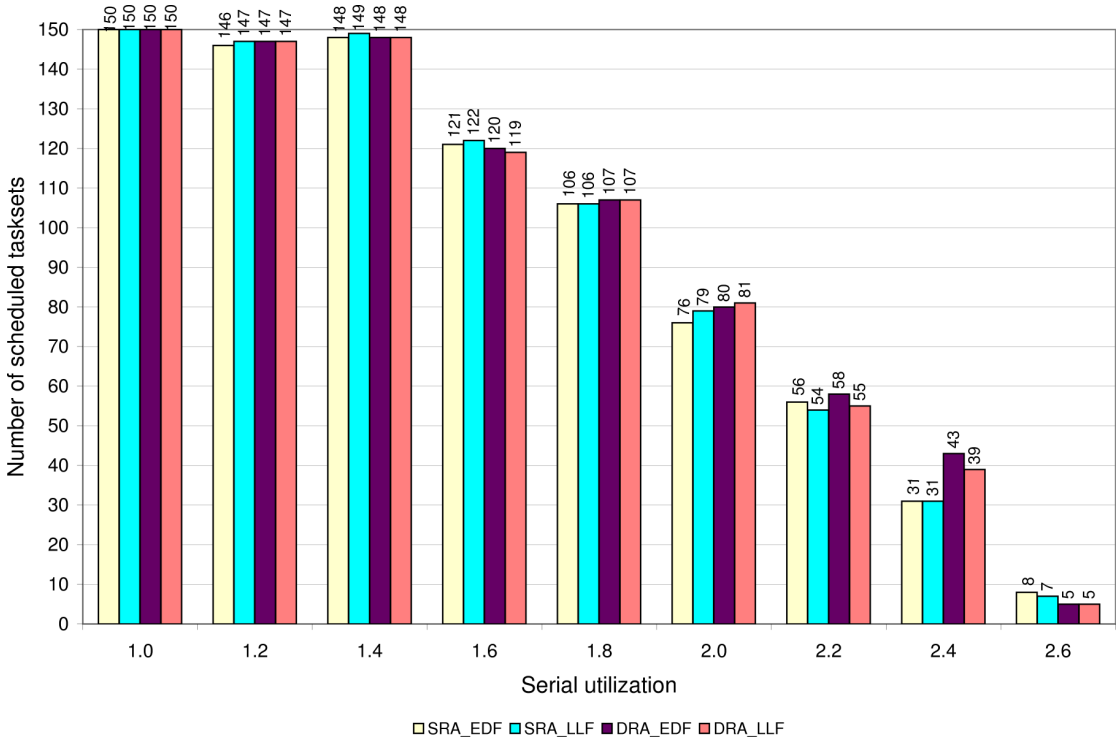


FIGURE 6.10: Comparison between our proposed algorithms (4-,8-,12-tasks).

Putting all together, Figure 6.10 shows all our proposed algorithms. They schedule very similar number of tasks, which highlights the fact that the biggest improvement comes from coupling the internal processor's resource allocation to the software targets. The DRA versions tend to outperform by a small margin the SRA versions, among them, SRA-EDF is the scheduler that presents the smallest performance improvement (11.8% when compare do EDF). Nevertheless, we expect the choice between our proposals

[5]Recall that we consider a task set successfully scheduled when there is no missed deadline.

to depend on more than their comparative performance. For instance, LLF based algorithms increase the scheduling complexity due to the fact that frequent laxity calculation must be performed for all tasks. On the other hand, DRA based algorithms require a more complex hardware support that may not be available.

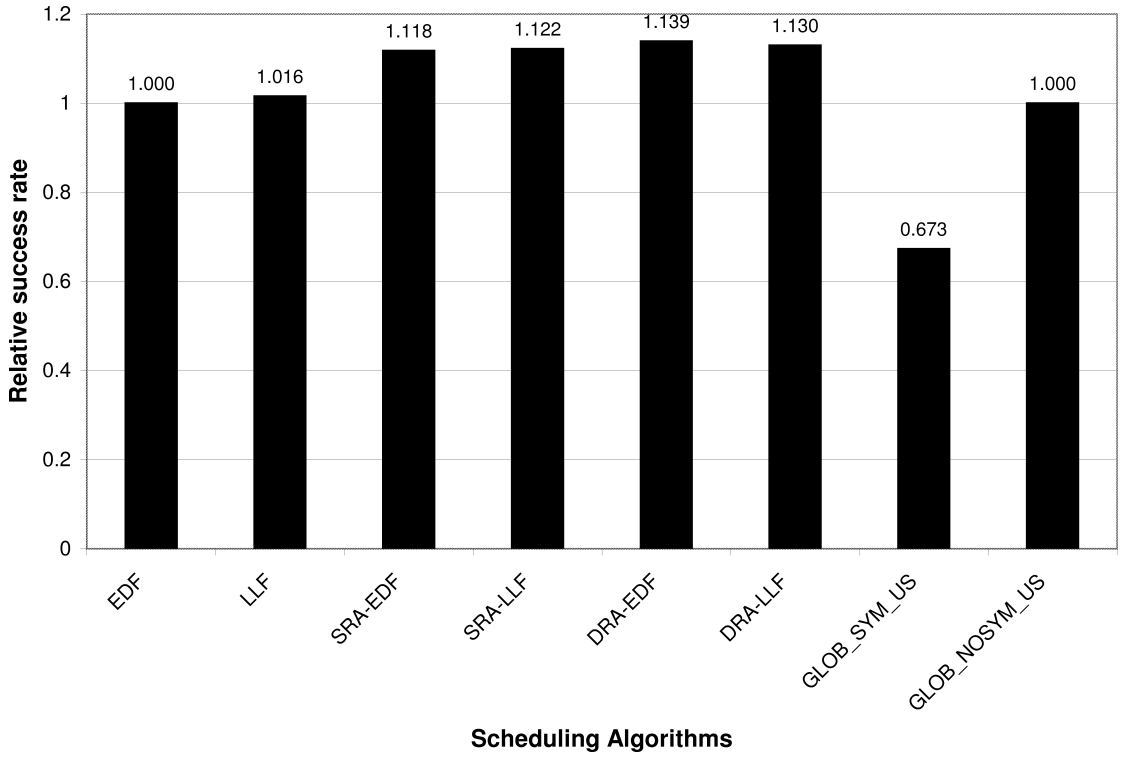


FIGURE 6.11: Aggregated success rate for all serial utilizations normalized to EDF.

Figure 6.11 shows the number of scheduled task sets, normalized to EDF, for all serial utilizations and all schedulers. As we can see, the benchmark that performs the worse in our environment is the GLOB_SYM_US, scheduling 32.7% less task sets than EDF. The best aggregated result is the one obtained by DRA-EDF, which is very similar to the one obtained by DRA-LLF (0.9% difference).

We should observe that, except for one task set, with serial utilization of 1.6, where DRA-EDF and DRA-LLF miss, there is no case in which our proposed algorithm has lower success rate than any of the others. Furthermore, using this resource aware scheduling algorithms eliminates the hardware-software priority inversion problems. That is, in contrast to normal SMT processors, where the hardware scheduler (fetch priority mechanism) and the OS scheduler are not aware of each other, there is no case where a lesser priority thread consumes more resources than the higher priority one. In addition, our solutions do not require profiling of the tasks to schedule.

Another key observation is the fact that we used a very aggressive WCET estimate. The closer is the WCET to the average execution time, the harder will it be to schedule the task set, as there will be close to zero extra time on normal execution. On real systems, WCET are normally an upper bound on the execution time, and, therefore, even more task sets would be scheduled on the common case. Considering a soft real-time scenario, using this slack between the WCET and the "expected execution time" would be, on some cases, an acceptable situation.

From our proposals, whenever the necessary hardware support can be implemented, we estimate that DRA-EDF should be chosen. Otherwise, SRA-EDF should be preferred. It presents very similar results to SRA-LLF and much lower complexity.

6.6 Conclusions

Embedded systems require increasingly high throughput rates. To reach those rates, current embedded processors use features that resemble to the ones used in the high-performance processors. The use of these features impacts the performance predictability and creates new problems for real-time system. SMT processors are a clear example of this new trend. SMTs provide higher throughput with reduced costs but make harder the problem of computing the worst case execution time, generating task interference or even giving most of the shared hardware resources to a task with lower priority when multiple tasks are running on different hardware threads.

In this chapter, we address the problem of scheduling a task set in a soft real-time SMT system from the software and hardware layers in a collaborative way. Our proposal allows better control of the underlying hardware resources by the scheduling algorithm, increasing the task scheduling success rate. Assuming that the Worst Case Execution Time (WCET) is given, for every task, our mechanisms do not require any additional profiling.

The original Earliest Deadline First (EDF) algorithm, used in most RT systems, only aims to determine the order in which tasks should be executed. This is not enough if the task set is scheduled on an SMT processor due to the execution time variability of threads. We developed and evaluated, in a simulated environment, a new scheduling algorithm, called RA-EDF (Resource-Aware EDF), that uses a hardware support proposed in [16]. RA-EDF, in addition to determine the execution order of threads, controls the amount of resources to give to co-scheduled threads. We provide EDF with knowledge of the processor resources and instruct it in how to split resources among threads

in order to meet their deadlines. This increases the success rate when scheduling tasks, outperforming state-of-the-art scheduling algorithms. The proposed scheduler algorithm obtains better results than EDF on every case when compared to the previous proposed task schedulers: 11.8% average improvement.

In addition, we developed RA-LLF, a resource-aware LLF variant that, in addition to control the resource allocation, improves the laxity accounting. RA-LLF achieves better results than RA-EDF when there is skewed distribution of utilization between the tasks. Furthermore, we propose a new hardware support that allows hardware-level fine-grain dynamic prioritization. The RA-EDF and RA-LLF versions adapted to the dynamic prioritization hardware were called DRA-EDF and DRA-LLF respectively. DRA-EDF and DRA-LLF achieve better results than RA-EDF and RA-LLF. As a future work, we plan to expand the proposed mechanisms to an N-way SMT machine: keeping one MCT and many LPT would make it feasible. Furthermore, we would like to evaluate other secondary metrics for the double-objective hardware.

Chapter 7

Conclusions

Multithreaded processors became widely used in academia and industry as a way to increase the aggregated performance (throughput). They offer additional performance at a low cost and low complexity. Furthermore, they allow to increase the processor's throughput with relatively small power increase.

Due to their resource sharing, these processors present specific problems and characteristics. Hardware-software priority inversion (by the hardware resource allocation) and performance variability are two examples of such problems. These problems are caused or worsened by the fact that several separated hardware policies rule the allocation of the many levels of resource sharing: the cache replacement policy is not coordinated with the internal resource sharing policy or the fetch policy and so on. And even worse, the operating system or the software layer has no way to enforce the software targets.

In this thesis, we use a comprehensive approach, coordinating targets of software and hardware. We employ hardware with explicit resource allocation and propose new hardware support when such hardware is not commercially available. For both the new simulated hardware and ones commercially available, we propose new software-controlled mechanisms to narrow the gap between software and hardware policies.

The main contributions of this thesis are:

1. On the real time domain, we addressed the problem of scheduling a task set in a SMT system from the software and hardware layers in a collaborative way. We made two sets of contributions: scheduling mechanisms that improve the schedulability of the previous proposed soft real-time task schedulers, and an improved hardware support for explicit resource allocation that allows targeting a given metric while still guaranteeing a minimum performance for the high priority task.

The scheduling algorithms, RA-EDF and RA-LLF, when compared to EDF and LLF, improved success ratio scheduling soft real-time tasks by 11.8% and 10.5%, respectively. When running with the new hardware support, another 2.1% are harvested for DRA-EDF, improving EDF by 13.9%. DRA-LLF improves LLF by 11.2%.

2. On the HPC domain, we made the following contributions:

   - We characterized a real processor, the IBM POWER5, and perform an in-depth evaluation of the effects of this processor's software-controlled prioritization mechanism over several different loads.

     We present the following conclusions: first, workloads presenting a large amount of long-latency operations are less influenced by priorities then the ones executing low-latency operations (i.e., integer arithmetic). Second, it is possible, by using the prioritization mechanism, to improve the overall throughput up to two times, in very special cases. However, those extreme improvements often imply drastic reduction of the low IPC thread's performance. On less extreme cases, it is possible to improve the throughput by 40%. And third, we show that, instead of using the full spectrum of priorities, only priorities up to +/-2 should be used, while "extreme" priorities should be used only when the performance of one of the two threads is not important.

   - We propose the use of the thread prioritization mechanism as a way to perform load-balance in HPC applications and significantly reduce their execution time.

   - We propose HPCSched, a scheduler for the Linux kernel that is able to perform load-balancing transparently and dynamically, adapting to application's behavior changes and requiring no changes in the program code.

   - We propose DLRB, a transparent dynamic user-level load and resource balancer in the form of a linked library.

     Our experiments show that HPCSched and DLRB achieve results close to the one obtained by static hand-tuning of the priorities for applications with constant behavior. In addition, for the cases where the static hand tuning cannot perform well due to the fact that the applications exhibit dramatic changes in behavior, both HPCSched and DLRB are able to adapt and re-balance the resources appropriately.

Summing up our findings, we conclude that, by a coordinated hardware/software approach in which the system software and the hardware tightly collaborate, it is possible to develop improved heuristics to target different metrics. In this thesis, we control, from the software, the hardware resource allocation to improve two of the possible

metrics: the resource allocation to improve the schedulability of tasks in soft real-time systems and the resource balancing as a way to reduce applications execution time in high performance computing.

7.1 Future work

This thesis opened several new topics that we want to better explore. Among others, we would like to highlight the following:

- In the real time systems, we would like to explore mechanisms that allow predictable performance and improved scheduling in new hardware with more than two hardware threads and/or with multiple cores.
- In HPC systems, we are now working on the combined use of hardware explicit resource allocation and additional actuation mechanisms (for instance, dynamic voltage scaling or coscheduling) to improve additional metrics, like power or energy. In fact, the prioritization may be an alternative to situations when dynamic voltage scaling is not desirable or too coarse grain. Our current work targets a scheduler that is able to use several actuation mechanisms coordinated at the same time.
- Also, for HPC systems, we plan to expand our load-balancing solutions to a cluster level: HPCSched and DLRB are able to balance HPC application inside a node but modern Supercomputers consists of Thousands of nodes. In this case there is another level of load balancing which consists of assigning the correct group of tasks to each node considering that the local scheduler is able to dynamically assign more or less hardware resource to each task.

Some of these topics are already being developed. We hope to deal with the remaining topics in the near future.

Appendix A

Published work

A.1 Conferences

- Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla and Mateo Valero. A Dynamic Scheduler for Balancing HPC Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Austin, USA. November 15-21, 2008.

- Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Chen-Yong Cher, Alper Buyuktosunoglu and Mateo Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *International Symposium on Computer Architecture (ISCA)*. Beijing, China. June 21-25, 2008.

- Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Julita Corbalan, Jesus Labarta and Mateo Valero. Balancing HPC Applications Through Smart Allocation of Resources in MT Processors. In *International Parallel & Distributed Processing Symposium (IPDPS)*. Miami, Florida, USA. April 14-18, 2008.

- Carlos Boneti, Francisco J. Cazorla and Mateo Valero. *Improving EDF for SMT processors. XVII Jornadas de Paralelismo (JP06)*. Albacete, Spain. September 2006.

A.2 Journals

- Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa and Mateo Valero. Soft Real-Time Scheduling on SMT Processors with Explicit Resource Allocation. In

International Conference on Architecture of Computing Systems (ARCS). Dresden, Germany. February 25,2008. Lecture Notes in Computer Science. Volume 4934/2008

A.3 Book Chapters

- Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla and Mateo Valero. Using resource allocation to balance HPC applications. To appear in: *Parallel and Distributed Computing*, IN-TECH, Viena, Austria ISBN978-3-902613-45-5, 2009.

A.4 Workshops and Poster Abstracts

- Carlos Boneti, Carlos Villavieja, Isaac Gelado, Marisa Gil and Nacho Navarro. Scheduling techniques for SMT processors. In *International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2005)*, L'Aquila, Italy, July 2005. Academic Press, ISBN 90 382 0802 2.

- I. Gelado, C. Villavieja, C. Boneti, M. Gil, X. Martorell. "Distributed Resource Management". In *International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2005)*, L'Aquila, Italy, July 2005. Academic Press, ISBN 90 382 0802 2.

- C. Villavieja, I. Gelado, C. Boneti, M. Gil, N.Navarro. "Runtime Power Consumption measurements in Wireless Sensor Networks". In *International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2005)*, L'Aquila, Italy, July 2005. Academic Press, ISBN 90 382 0802 2.

A.5 Technical Reports

- C. Boneti, F. J. Cazorla, M. Valero. Scheduling Improvements for Real-Time SMT Systems. Technical Report UPC-DAC-RR-2006-16.

Bibliography

[1] Metis - family of multilevel partitioning algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis.

[2] Siesta: A linear-scaling density-functional method. http://www.uam.es/siesta/.

[3] The Message Passing Interface (MPI) Standard. http://www.mcs.anl.gov/research/projects/mpi/.

[4] The TOP500 Supercomputing Sites, June 2007. http://www.top500.org/lists/2007/06.

[5] D. Alpert. Will microprocessor become simpler? *Microprocessor Report*, Nov 2003.

[6] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, Eric Rotenberg, and Frank Mueller. Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*, volume 31, 2 of *Computer Architecture News*, pages 350–361, New York, June 9–11 2003. ACM Press.

[7] Eduard Ayguade, Bob Blainey, Alejandro Duran, Jesus Labarta, Francisco Martinez, Xavier Martorell, and Raul Silvera. Is the schedule clause really necessary in openMP? In *Proceedings of the 4th International Workshop on OpenMP Applications and Tools, (WOMPAT'03)*, volume 2716 of *Lecture Notes in Computer Science (LNCS)*, pages 147–159, Toronto, Canada, June 2003. Springer-Verlag (New York).

[8] Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, Chen-Yong Cher, and Mateo Valero. Software-controlled priority characterization of POWER5 processor. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08)*, Beijing, June 2008. ACM SIGARCH.

[9] Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, Julita Corbalan, Jesus Labarta, and Mateo Valero. Balancing HPC applications through smart allocation of resources in MT processors. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium (IPDPS'08)*, pages 1–12. IEEE, 2008.

[10] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly & Associates, Inc., 3rd edition, 2006.

[11] Jeffery A. Brown and Dean M. Tullsen. The shared-thread multiprocessor. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS'08)*, pages 73–82, New York, NY, USA, 2008. ACM Press.

[12] Francisco J. Cazorla, Enrique Fernandez, Peter M. W. Knijnenburg, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. On the problem of minimizing workload execution time in SMT processors. In *Proceedings of 2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS'07)*, pages 66–73. IEEE, 2007.

[13] Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Improving memory latency aware fetch policies for SMT processors. In *Proceedings of the 5th International Symposium on High Performance Computing (ISHPC'03)*, volume 2858 of *Lecture Notes in Computer Science*, pages 70–85. Springer, October 2003.

[14] Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez, and Mateo Valero. DCache Warn: an I-Fetch policy to increase SMT efficiency. In *Proceedings of the 18th Internation Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 74–, Los Alamitos, CA, USA, April 2004. IEEE Computer Society.

[15] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Implicit vs. explicit resource allocation in SMT processors. In *Proceedings of the EUROMICRO Systems on Digital System Design (DSD'04)*, pages 44–51, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Architectural support for real-time task scheduling in SMT processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'05)*, pages 166–176. ACM Press, 2005.

[17] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, 2006.

[18] Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Predictable performance in SMT processors. In *Proceedings of the First Conference on Computing Frontiers (CF'04)*, pages 433–443, New York, NY, USA, 2004. ACM Press.

[19] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*, pages 171–182, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, Peter M. W. Knijnenburg, Rizos Sakellariou, and Enrique Fernandez. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, 24(4):24–31, July/August 2004.

[21] Onur Celebioglu, Amina Saify, Tau Leng, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. The performance impact of computational efficiency on HPC clusters with hyper-threading technology. In *Proceedings of the 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'04)*, Santa Fe, New Mexico, USA, April 2004. IEEE Computer Society (Los Alamitos, CA).

[22] Derek Chiou, Larry Rudolph, Srinivas Devadas, and Boon S. Ang. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference (DAC'00)*, June 2000.

[23] Seungryul Choi and Donald Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings 33th International Symposium on Computer Architecture (ISCA'06)*, volume 34, pages 239–251, Boston, MA, USA, June 2006. ACM SIGARCH / IEEE Computer Society.

[24] Matthew Curtis-Maury and Tanping Wang. Integrating multiple forms of multithreaded execution on multi-SMT systems: A study with scientific applications. In *Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 199–209, Torino, Italy, 2005. IEEE Computer Society.

[25] Ronaldo de Lara Goncalves and Philippe Olivier Alexandre Navaux. Improving SMT performance scheduling processes. In *Proceedings of the 10th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'02)*, volume 0, pages 327–334, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[26] Michael L. Dertouzos and Aloysius K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989.

[27] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, January 1978.

[28] Gautham K. Dorai and Donald Yeung. Transparent threads: Resource sharing in SMT processors for high single-thread performance. *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, pages 30–41, Sep 2002.

[29] Alejandro Duran, Marc Gonzalez, Julita Corbalan, Xavier Martorell, Eduard Ayguade, Jesus Labarta, and Raul Silvera. Automatic thread distribution for nested parallelism in OpenMP. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS'05)*, pages 121–130, Cambridge, Massachusetts, USA, June 2005.

[30] Ali El-Haj-Mahmoud, Ahmed S. Al-Zawawi, Aravindh Anantaraman, and Eric Rotenberg. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'05)*, pages 213–224. ACM Press, 2005.

[31] Ali El-Haj-Mahmoud and Eric Rotenberg. Safely exploiting multithreaded processors to tolerate memory latency in real-time systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*, pages 2–13. ACM Press, 2004.

[32] Ali El-Moursy and David H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th International Conference on High Performance Computer Architecture (HPCA'03)*, pages 31–40, Anaheim, California, USA, February 2003. IEEE Computer Society.

[33] Felix Freitag, Julita Corbalan, and Jesus Labarta. A dynamic periodicity detector: Application to speedup computation. In *Proceedings of the 15th the International Parallel and Distributed Processing Symposium (IPDPS'01)*, pages 2–2, Los Alamitos, CA, 2001. IEEE Computer Society.

[34] Isaac Gelado, Javier Cabezas, Lluis Vilanova, and Nacho Navarro. The cost of IPC: an architectural analysis. In *Proceedings of the International Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA'07)*. ACM Press, 2007.

[35] Ben Gibbs, Balaji Atyam, Frank Berres, Bruno Blanchard, Lancelot Castillo, Pedro Coelho, Nicolas Guerin, Lei Liu, Cesar Diniz Maciel, Carlos Sosa, and Ravikiran Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. Redbook. IBM, International Technical Support Organization, Austin, TX, USA, 2005.

[36] Roberto Gioiosa, Fabrizio Petrini, Kei Davis, and Fabien Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Proceedings of the 4th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'04)*, pages 387–390, Rome, Italy, 2004.

[37] Joel Goossens and Christophe Macq. Limitation of the hyper-period in real-time periodic task set generation. In *Proceedings of the RTS Embedded Systems (RTS'01)*, pages 133–148, Paris, France, 2001.

[38] Joel Goossens and Pascal Richard. Overview of real-time scheduling problem. In *Proceedings of the 9th International Conference on Project Management and Scheduling*, pages 13–22, Nancy France, April.

[39] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[40] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO'07)*, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[41] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4), 2006.

[42] Weicheng Huang and Danesh Tafti. A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. In *Proceedings of the Parallel Computational Fluid Dynamics (PCFD'99)*, 1999.

[43] IBM. *PowerPC Operating Environment Architecture version 2.02*. Number 3 in PowerPC Architecture books. 2005.

[44] IBM. Cell broadband engine programming handbook v1.11, 2008.

[45] IBM, Sony, and Toshiba. Cell broadband engine architecture v1.01, 2006.

[46] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa R. Hsu, and Steve K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*, pages 25–36, New York, NY, USA, 2007. ACM Press.

[47] Rohit Jain, Christopher J. Hughes, and Sarita V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 134, Washington, DC, USA, December 2002. IEEE Computer Society.

[48] Haoqiang Jin and Rob F. Van der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. *Journal of Parallel and Distributed Computing*, 66(5):674–685, 2006.

[49] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. SMT implementation in POWER5. In *Hot Chips*, volume 15, Aug 2003.

[50] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM POWER5 Chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[51] Peter M. W. Knijnenburg, Alex Ramirez, Josep L. Larriba-Pey, and Mateo Valero. Branch classification for SMT fetch gating. In *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation (MTEAC'02)*, pages 49–56, November 13 2002.

[52] Jesus Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A parallel program development environment. In *Proceedings of the 2nd International Conference on Parallel Processing (Euro-Par'96)*, volume II of *Lecture Notes in Computer Science*, pages 665–674, Lyon, France, 1996. Springer.

[53] Hung Q. Le, William J. Starke, J. Stephen Fields, Francis P. O'Connell, Dung Q. Nguyen, Bruce J. Ronchetti, Wolfram Sauer, Eric M. Schwarz, and Michael T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[54] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO'97)*, pages 330–335, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society.

[55] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D geometry transformations on a simultaneous multithreaded SIMD processor. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*, pages 236–245, Sorrento, Napoli, Italy, May 2001. ACM.

[56] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[57] Shi-Wu Lo, Kam-Yiu Lam, and Tei-Wei Kuo. Real-time task scheduling for SMT systems. In *Proceedings of the International Workshop on Real-Time Computing Systems and Applications (RTCSA'05)*, pages 5–10, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[58] Kun Luo, Manoj Franklin, Shubhendu S. Mukherjee, and Andre Seznec. Boosting smt performance by speculation control. In *Proceedings of the 15th the International Parallel and Distributed Processing Symposium (IPDPS'01)*, page 2, Los Alamitos, CA, 2001. IEEE Computer Society.

[59] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'01)*, pages 164–171. IEEE, November 2001.

[60] Paul Mackerras, Thomas S. Mathews, and Randal C. Swanberg. Operating system exploitation of the POWER5 system. *IBM Journal of Research and Development*, 49(4/5):533–539, July 2005.

[61] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, James Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[62] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. MLP-aware dynamic cache partitioning. In *Proceedins of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'08)*, volume 4917 of *Lecture Notes in Computer Science*, pages 337–352, Goteborg, Sweden, January 2008. Springer.

[63] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th International Symposium on Microarchitecture (MICRO'07)*, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society.

[64] NASA. NAS parallel benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[65] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8, 192 processors of ASCI Q. In *International Conference for High Perfromance Computing, Networking, Storage and Analysis (SC'03)*, page 55. IEEE/ACM SIGARCH, 2003.

[66] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE Computer Society, 2006.

[67] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 6th International Conference on Parallel Processing (Euro-Par'00)*, volume 1900 of *LNCS*, pages 296–310. Springer-Verlag, Berlin, 2000.

[68] Mauricio J. Serrano, Roger C. Wood, and Mario Nemirovsky. A study on multistreamed superscalar processors. In *Technical Report 93-05, University of California Santa Barbara*, 1993.

[69] Balaram Sinharoy, Ronald N. Kalla, Joel M. Tendler, Richard J. Eickemeyer, and Jody B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[70] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *ACM SIGPLAN Notices*, 35(11):234–244, November 2000.

[71] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*, volume 30 of *SIGMETRICS Performance Evaluation Review*, pages 66–76, New York, June 15–19 2002. ACM Press.

[72] Jose M. Soler, Emilio Artacho, Julian D. Gale, Alberto Garcia, Javier Junquera, Pablo Ordejon, and Daniel Sanchez-Portal. The SIESTA method for ab initio order-n materials simulation. *Journal of Physics: Condensed Matter*, 14(11), 2002.

[73] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

[74] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th International Conference on Supercomputing (ICS '05)*, pages 303–312, New York, NY, USA, 2005. ACM Press.

[75] Dean M. Tullsen and Jeffery A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO'01)*, pages 318–327, Austin, Texas, December 1–5, 2001. IEEE Computer Society.

[76] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack. L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)*, pages 191–202, New York, NY, USA, 1996. ACM Press.

[77] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 392–403, June 22–24 1995.

[78] Javier Vera, Francisco J. Cazorla, Alex Pajuelo, Oliverio J. Santana, Enrique Fernandez, and Mateo Valero. FAME: Fairly measuring multithreaded architectures. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT'07)*, pages 305–316. IEEE Computer Society, 2007.

[79] Javier Vera, Francisco J. Cazorla, Alex Pajuelo, Oliverio J. Santana, Enrique Fernandez, and Mateo Valero. Measuring the performance of multithreaded processors. In *SPEC Benchmark Workshop*, 2007.

[80] Chris H. Walshaw and Mark Cross. Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes. In *Computational Mechanics Using High Performance Computing*, pages 79–94. Saxe-Coburg Publications, Stirling, 2002.