CPU ACCOUNTING IN MULTI-THREADED PROCESSORS

José Carlos Ruiz Luque
(Carlos Luque)

Barcelona, 2014

A thesis submitted in fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Department of Computer Architecture
Universitat Politècnica de Catalunya

CPU ACCOUNTING IN MULTI-THREADED PROCESSORS

José Carlos Ruiz Luque

(Carlos Luque)

Barcelona, 2014

Thesis director:

Francisco J. Cazorla Almeida
Barcelona Supercomputing Center
Spanish National Research Council
Universitat Politècnica de Catalunya

Thesis co-directors:

Miquel Moretó Planas
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Mateo Valero Cortés
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

A thesis submitted in fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Department of Computer Architecture
Universitat Politècnica de Catalunya

A mi hermano *Roberto* y
mi madre *Dori*

Abstract

In recent years, multi-threaded processors popularity has raised in industry in order to increase the system aggregated performance and per-application performance, overcoming the limitations imposed by the limited instruction-level parallelism, and by power and thermal constraints.

However, multi-threaded processors introduce complexities when accounting CPU (computation) capacity (CPU accounting), since the CPU capacity accounted to an application not only depends upon the time that the application is scheduled onto a CPU, but also on the amount of hardware resources it receives during that period. And given that in a multi-threaded processor hardware resources are dynamically shared between applications, the CPU capacity accounted to an application in these processors depends upon the workload in which it executes. This is inconvenient because the CPU accounting of the same application with the same input data set may be accounted significantly different depending upon the workload in which it executes. Deploying systems with accurate CPU accounting mechanisms is necessary to increase fairness among running applications. Moreover, it will allow users to be fairly charged on a shared data center, facilitating server consolidation in future systems.

This Thesis analyses the concepts of CPU capacity and CPU accounting for multi-threaded processors. This study demonstrates that current CPU accounting mechanisms are not as accurate as they should be in multi-threaded processors. For this reason, we present two novel CPU accounting mechanisms that improve the accuracy in measuring the CPU capacity for multi-threaded processors with low hardware overhead. We focus our attention on several current multi-threaded processors, including chip multiprocessors and simultaneous multithreading processors. Finally, we analyse the impact of shared resources in multi-threaded processors in operating system CPU scheduler and we propose several schedulers that improve the knowledge of shared hardware resources at the software level.

Acknowledgements

It is a great pleasure to find oneself at this stage of research life when the real adventure with science just begins, yet already with a clear taste of it. It is even a greater pleasure to be able to thank to all those that have contributed in some way to the completion of this Thesis.

In the first place, I would like to express my sincere gratitude to my advisor Francisco Javier Cazorla and my co-advisors Miquel Moretó and Mateo Valero for their dedication, encouragement and guidance throughout my Ph.D. adventure. Also, their criticism and encouragement has been a great inspiration for me. I have enjoyed every single discussion with them and I believe it will continue in the future. I will always be thankful for their valuable comments and remarkable patience. It has been an honour to be their Ph.D. student. Thanks to them I have learnt what *research* truly means!

I am also greatly indebted to Enriquez Fernández for his excellent guidance at the early stage of my career. He introduced me to the computer architecture. I would also like to thank my collaborators, Roberto Gioiosa, Alper Buyuktosunoglu and Alexandra Fedorova for great interactions and endless discussions.

Special thanks go to my colleagues from the group 'CAOS' in Barcelona Supercomputing Center for an overwhelming support and a great encouragement at the very last stage of this Thesis.

This work would not have been possible without the financial support of Barcelona Supercomputing Center, Universitat Politècnica de Catalunya, Generalitat de Catalunya and the Spanish Ministry of Education and Science.

Last but not least, I wish to thank my Family for their love, for patience, and for their understanding and gigantic encouragement through difficult moments. If it had not been them, this Thesis would not come into existence.

Contents

Chapter 1
Introduction

Over the last 20 years, the performance of Single-Threaded (ST) processors has increased due to two factors: *advances in the integrated circuit technology* and *microarchitectural techniques* such as deeper pipeline, dynamic scheduler, speculative execution through branch predictor, cache hierarchy and non-blocking caches. Advances in the integrated circuit technology have provided both faster and smaller transistors. Smaller transistors increase integration while faster transistors allow increasing clock rates, and hence, processors can run at higher clock rate, in other words, high frequency. Microarchitectural techniques allow exploiting Instruction-Level Parallelism (ILP) inherent in tasks[1]. Although microarchitectural techniques help improving performance, a large portion of this performance comes from advances in integrated circuits. On the whole, processors run instructions very fast due to high frequency and the number of instructions running concurrently is high due to microarchitectural techniques.

Due to power and thermal constraints [20] and the limited amounts of ILP inherent in tasks [66], new techniques have been developed to improve performance with low energy requirements. One such technique is Thread-Level Parallelism (TLP). Current processors implement at least one form of TLP, which allows executing several tasks onto the processor simultaneously.

TLP can be implemented across multiple chips. For instance, Symmetric MultiProcessing (SMP) consists of two or more processors that are connected to shared hardware resources such as main memory. TLP can also be implemented at the chip level. In this level, a wide variety of processors supports TLP. On one extreme of the spectrum, we find Simultaneous Multithreading (SMT) processors [52, 64], in which tasks share most of the processor resources. On the other end of the spectrum,

[1] In this Thesis, we use the term task to refer to an application running in a computing system.

Chip MultiProcessors (CMP) [42] or multi-core processors, in which tasks share only some levels of the cache hierarchy and the memory bandwidth. In between, there are other TLP paradigms like Coarse-Grain MultiThreading (CGMT) [2, 62] or Fine-Grain MultiThreading (FGMT) [19, 57]. Each of these designs offers different benefits as they exploit TLP in different ways, which motivates processor vendors to combine different TLP paradigms in their latest processors. Some notorious examples are the Intel core i7 [51] and IBM POWER7 [56], which are CMP+SMT processors, and the ORACLE UltraSPARC T4 [43] that is CMP+FGMT. This trend in integrating different TLP paradigms will keep growing in importance according to the ITRS roadmap for the future years [24]. In this Thesis, we make use of the term Multi-Threaded (MT) processor to refer to a processor implementing any TLP paradigm or combination of TLP paradigms at chip level.

Both SMP systems and MT processors permit to execute several tasks simultaneously. In this way, the utilization of the hardware resources is improved because several tasks can use different parts of hardware resources, and hence, the system throughput is increased.

On the other hand, the communication latency in SMPs is higher than in MT processors because on-chip communications are faster than outside the chips used in SMPs. For instance, if two or more threads of a parallel task want to share data, the communication latency will be lower in a MT processor as data will be cached on-chip (in the cache hierarchy), and not in main memory as in a SMP. The SMP can have a cache hierarchy with a Last level Cache (LLC) shared among processors, but this LLC is outside the chip. Consequently, increased communication latencies are obtained on those systems.

Other benefits of MT processors are the design costs and power efficiency. The design costs are reduced because the design complexity of a MT processor is fairly less than a processor with a larger monolithic core. Due to the described benefits of MT processors, processor vendors prefer this microarchitecture to improve the processor performance with low energy requirements.

1.1 Challenges for Accurate CPU Accounting in Multi-Threaded Processors

An Operating System (OS) is a software application that manages computer hardware resources and provides common services for computer systems. It also acts as an

interface between user and the computer hardware and supplies an environment within which user applications can do useful work [55]. Its major responsibility is to manage and ensure proper operation of the hardware resources. Moreover, an OS provides the user with an abstraction of the hardware resources. The user application perceives this abstraction as if it was using the complete hardware while, in fact, the OS shares hardware resources among the user applications. For example, several tasks demand service from the Central Processing Unit (CPU). Hardware resources can be shared *temporally* and *spatially*. Hardware resources are time shared between users when each task can make use of a resource for a limited amount of time (for example, the exclusive use of a CPU). Orthogonally, hardware resources can be shared spatially when each task makes use of a limited amount of resources, such as cache memory or I/O bandwidth.

In traditional, ST uniprocessor systems, the execution time of a task is influenced by the amount of hardware resources shared with the other running tasks. It is also affected by how long the task runs with other tasks. However, *the time accounted to that task is roughly the same regardless of the workload*[2] *in which it is executed, in other words, regardless of how many tasks are sharing the hardware resources at any given time*. We call this principle, the *Principle of Accounting*.

To illustrate this principle of accounting, we use an Unix-like system that differentiates the real execution time and the time a task actually is running onto a CPU. Commands such as `time` or `top` provide three values: *real*, *user* and *sys*. *Real* is the elapsed wall clock time between the invocation and termination of the task; *user* is the time spent by the task in the *user mode*; and *sys* is the time spent in the *kernel mode* on behalf of the task. In these systems, *sys+user* time is the execution time accounted to a task.

Figure 1.1 shows the total (*real*) and the accounted execution time (*sys+user*) of the 171.*swim* (or simply *swim*) SPEC CPU 2000 benchmark [60] when it runs in different workloads. In this figure, the time results are normalized to the real execution time of *swim* when it runs in isolation[3], meaning that, once *swim* is scheduled on a CPU it does not share the CPU resources with any other task. For this experiment, we use an Intel Xeon Quad-Core processor at 2.5 GHz (though the general trends drawn from Figure 1 apply to other current MT processor). There are four cores in the chip,

[2] In this Thesis, we use the term workload to refer to a set of tasks running onto a processor simultaneously.

[3] In this Thesis, we use the term execution in *isolation* to refer to when a task runs alone onto a processor.

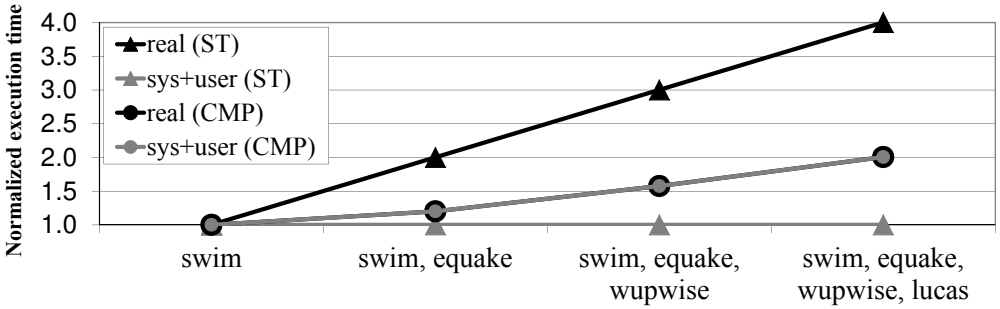


Figure 1.1: Total (*real*) and accounted (*sys+user*) time of *swim* in different workloads running on an Intel Xeon Quad-Core CPU

on which we run Linux 2.6.18. We move most of the OS activity to the first core, leaving the other cores as isolated as possible from *OS noise*. When *swim* runs alone in one of the isolated cores, it completes its execution in 117 seconds. However, when *swim* runs together with other tasks in the same core, its real execution time increases up to 4x due to context switches forced by the OS (black triangles in Figure 1.1). Nevertheless, *swim* is accounted roughly the same time (grey triangles), which is the time the task actually uses the CPU. Tasks may suffer some delay because of cache data eviction and the loss of TLB contents caused by a context switch, but this effect is small in this case. Hence, even if total execution time of *swim* increases depending upon the other tasks it is co-scheduled with, the time accounted to *swim* is always the same.

In ST uniprocessor systems, each running task uses 100% of the processor's resources and its progress can be generally measured in terms of the time spent running onto the CPU. We call this CPU accounting mechanism the *Classical Approach*. The classical approach has been proved to work well for ST uniprocessor and SMP systems, as the amount of hardware resources spatially shared is limited.

However, MT processors make more complex to measure CPU utilization because the progress of a task depends upon the activity of the other tasks running at the same time. Current CPU accounting mechanisms may lead to inaccuracy for the time accounted to each task. In order to show this inaccuracy, in a second experiment, we use all the cores in the Intel Xeon Quad-Core processor. We execute *swim* concurrently with other tasks, as shown by the x-axis in Figure 1.1. In this case, *swim* suffers no time sharing and *real* time is roughly the same as *sys+user* because the number of tasks that are running is equal or lower than the number of virtual CPUs (see Figure 3.1) in the system. In Figure 1.1, the grey circles show a variance up of to 2x in the

time *swim* is accounted for depending upon the workload in which it runs. This means that *a task running onto a MT processor may be accounted different CPU utilization depending upon the other tasks running on the same chip at the same time*. From the user point of view this is an undesirable situation, as the same task with the same input data set is accounted differently depending upon the tasks it is co-scheduled with.

In short, the principle of accounting is not guaranteed in MT processors due to the fact that the performance of a task depends on both the time the task runs onto the processor and the amount of resources it receives during that time. The latter is in general not under the control of the user or the OS. To make things worse, there is a non-linear relation between the percentage of resources assigned to a task and the slowdown it suffers with respect to running in isolation.

The inaccuracy measuring per-task CPU utilization may affect several key components of a computing system, such as performance monitoring tools, the CPU scheduler, or billing mechanisms in data centers:

- *Performance tools* generate statistics of various parameters of a task such as CPU utilization and I/O activities in order to know the behaviour of the task in a computing system. For example, in Unix-like systems the command *vmstat* reports statistics regarding virtual memory, disks, and CPU activity. These tools may not properly account the CPU utilization of task in a MT processor and hence, the tools generate wrong profile of tasks. This is undesirable. For instance, in a queue system, a task may exceed the threshold for the particular queue where it runs because its execution time changes depending upon who are its co-runners.

- In modern time-sharing OSes, the *CPU scheduling algorithm* multiplexes the CPU time for different tasks in the system and maintains fairness between tasks. In this scenario, the task schedulers assume that the time a task is scheduled onto CPU is a good metric in order to measure the CPU utilization of each task in ST uniprocessor system. However, in MT processors, this metric is flawed because each task may make a different progress in different workloads. For this reason, even if the CPU scheduler can balance time among tasks in the system achieving fairness, the scheduling algorithms cannot guarantee that a task progresses according to its software-assigned priority.

- *Selection of appropriate co-runners*. Task interaction in hardware shared resources may negatively affect tasks, hurting performance and increasing energy

requirements. Per-task CPU accounting can help the OS scheduler or a runtime-based scheduler to decide which tasks must be run and when, thus increasing the total system performance.

- *Billing in data centers*. Data centers charge users for the use of their resources. The fact that CPU utilization in MT processors is not accurately measured, makes current billing systems unfair when several users are consolidated onto the same processor. Measuring the CPU utilization each task performs, would allow charging each user more fairly. Traditionally, data centers offer services of compute capacity and storage to execution and store of its customers' tasks and data. In order to charge compute capacity to a customer, the data center may utilize the CPU utilization as measurement of the CPU activity. The payment of these services depends upon the utilized resources such as used amount of memory, disk space, I/O and CPUs activity, etc. For example, *Amazon Elastic Compute Cloud* (Amazon EC2) [3] is a virtual computing environment that leverages Amazon's own infrastructure to provide computing capacity to external users. The user sets up an image with all the required applications, libraries and data, and uploads them to Amazon's facilities. After this preliminary step, the user instructs Amazon EC2 to start executing an instance bound to previously uploaded image. At this point, Amazon starts billing the user for the number of hours the instance executes, for data transfers (network usage) and for disk space usage. In conclusion, having accurate per-task CPU utilization will facilitate server consolidation, allowing an efficient usage of the servers and a fair charging to users.

1.2 Thesis Contributions

This Thesis presents several proposals that effectively track the CPU capacity to account each task running onto MT processors. The contributions are divided into three groups. First, we formally define the CPU capacity and CPU accounting for MT processors. Second, at the hardware level, we propose two novel CPU accounting mechanisms that improve the accuracy in measuring the CPU accounting in MT processors. To start with, we study CPU accounting mechanisms for CMP processors. Next, CPU accounting mechanisms are studied for CMP+SMT processors, where each core supports SMT. Finally, at the software level, we study how the CPU scheduler should be adapted to take advantage of the proposed CPU accounting hardware mechanisms to

increase system fairness. The following sections briefly summarize each one of the contributions.

1.2.1 Concept of CPU Capacity and CPU Accounting for Multi-Threaded Processors

The first problem that is addressed in the Thesis is the definition of *CPU capacity* and *CPU accounting* for MT processors. Although these concepts are clearly defined in ST processors, they are vague in MT processors. For this reason, we provide two definitions of CPU capacity (*resource utilization* and *CPU progress*) and we also introduce the *fair share of CPU capacity* which represents the hardware resources that a task should get access when the task runs alone onto a MT processor. The fair share of CPU capacity can be defined with two different approaches: *full-share* and *even-share*.

In this Thesis, we focus on CPU capacity as CPU progress. The full-share approach assumes that a task running in isolation accesses all processor hardware resources during a given period of time. Consequently, the task execution progress of a task in a MT processor is equivalent to its execution progress when runs in isolation. Meanwhile, the even-share approach assumes that a task accesses even part of the processor resources. In our cases, even part of the processor resources is defined as $1/N$ of the processor resources, where N is the number of tasks supported by processors. In this way, the task execution progress of a task is equivalent to its execution progress when runs in isolation with an even part of the processor resources.

The *CPU accounting* is the process of measuring the CPU capacity utilized by task in a MT processor. In other words, the CPU accounting measures the CPU capacity of a task would have if the task run alone onto a MT processor. For instance, if a task A executes 5 milliseconds in a MT processor and the CPU accounting measures 4.5 milliseconds, it means that the task would run for 4.5 milliseconds if it is alone on the processor. The task progressed 90 percent of the time, and in the remaining time, the task was stalled due to interferences with other tasks. In contrast, in a ST processor, the task A would have progressed 100 percent of the time as it would run alone in the processor. We name *full-share accounting* when using CPU capacity based on CPU progress and full-share approach and *even-share accounting* when using CPU capacity based on CPU progress and even-share approach.

1.2.2 CPU Accounting for Multi-Core Processors

The second problem that we address in this Thesis is how the CPU accounting is obtained in CMP processors. CMP processors consist of simple processor cores and share only some levels of the cache hierarchy and the memory bandwidth (see Figure 2.1). We firstly focus on CMP processors because they have less shared hardware resource among tasks than SMT processors and we can easily identify the sources that generate inaccuracies when measuring CPU capacity in CMP processors with current CPU accounting mechanisms.

We propose a new CPU accounting mechanism for CMP processors, called *InterTask Conflict-Aware* (ITCA) accounting [33, 34]. The main goal of ITCA is to estimate accurately the CPU accounting to a task by keeping track of the interference in shared hardware resources, mainly in the on-chip shared cache. Moreover, we show that our understanding of the processor architecture in developing ITCA is correct by making a complete design space exploration of the possible combinations of hardware resource status indicator states. This exploration confirms that ITCA provides reasonably accurate results at moderate hardware cost. It also shows that with small changes we can implement an improved ITCA accounting mechanism, denoted I^2TCA, that further improves the accuracy of ITCA.

In addition, we show that ITCA works well with cache partitioning algorithms and that both *full-share* and *even-share* accounting approaches can be used with our proposed mechanisms.

1.2.3 CPU Accounting for CMP+SMT Processors

The third problem that is addressed in the Thesis is how the CPU capacity is measured in processors combining different TLP paradigms such as CMP+SMT processors. In these processors, more hardware resources are shared than CMP processors, since both on-core and off-core hardware resources are shared among running tasks. It is important to note that, in general, on-core shared resources in the SMT processor are more heavily shared than off-core resources shared among different cores.

ITCA does not adapt correctly to processors with multiple TLP paradigms because it cannot track interferences of shared on-core resources such as register files, issue queues, execution units, the reorder buffer, etc. In addition, the current CPU accounting mechanisms or combination of them either incurs an unaffordable hardware cost to track CPU capacity in processors with multiple TLP paradigms, or leads to inaccuracies in its measurement. Consequently, we introduce *Micro-Isolation Based*

Time Accounting (MIBTA) [35], a new CPU accounting mechanism to compute CPU capacity in CMP+SMT processors. Instead of adding hardware support in *each* shared hardware resource to track tasks' slowdown, MIBTA makes use of a time sampling technique in which tasks run in isolation for short periods of time, with negligible effect on the system throughput and with high accuracy in CPU accounting of tasks. In particular, our CPU accounting mechanism combines the following two mechanisms:

- At SMT level, where tracking how tasks interact in each core resource would introduce significant hardware overhead, our technique periodically runs each task in isolation to measure its CPU capacity. The execution of each task in a workload is divided into two phases that are executed in alternate fashion. In the first, *isolation* phase, all tasks but one are stopped so that the Instruction Per Cycle (IPC) in isolation of the task is measured. In a second, *multi-threaded* phase, all tasks run together. The ratio IPC_{MT}/IPC_{isol} times the total execution time gives the CPU accounting for each task. Since the number of available threads in each SMT processor is restricted (only 2-8 threads), this solution can be implemented with minimal performance degradation.

- At CMP level, our technique makes use of dedicated hardware monitoring support to track the interferences between tasks running in different cores. This hardware support is based on ITCA accounting mechanism [33, 34], which tracks the conflicts in the last level of cache, shared among all different cores, and estimates with high accuracy the CPU accounting in CMP processors. In addition, we propose a new monitoring hardware that significantly reduces the storage overhead of ITCA without affecting its high accuracy. The *Randomized Sampled Auxiliary tag directory*, denoted `RSA`, combines sampling techniques with randomized algorithms to predict inter-task misses to the entire LLC.

MIBTA combines both proposals to provide tight CPU accounting in CMP+SMT processors with reduced hardware overhead.

1.2.4 CPU Capacity-Aware Scheduling in Multi-Core Processors

Finally, we address how the software can be adapted to the proposed CPU accounting mechanisms in this Thesis with the aim of improving fairness. We focus on the CPU scheduler, which is a key component of the OS and has a significant impact in the system throughput and fairness.

The CPU scheduler focuses on efficiently sharing CPU resources among tasks. In ST uniprocessors, the CPU resources are shared temporarily, in other words, they are time-shared. The CPU scheduler selects the task that should be run next, and for how long this task should use the CPU resources. Time-based schedulers assume that the progress of a task is measured in terms of the time spent onto a CPU. On the other hand, for MT processors, which dynamically share a large amount of processor resources, this assumption is invalid because the task's progress depends upon the activity of other tasks in the same processor as illustrated in Figure 1.1.

In this Thesis, we define a new concept of *effective CPU capacity share* for MT processors, which is aware of the amount of actual CPU capacity used by a task during its execution. Moreover, we propose two novel task schedulers, *Balanced CPU Capacity Scheduler* and *Equal CPU Capacity Scheduler*, that are aware of the CPU capacity received by each running task, and make use of this information to share the CPU more effectively.

1.3 Thesis Structure

This Thesis is structured as follows:

- Chapter 2 focuses on explaining our experimental environment. This includes both the simulation tools and the benchmarks used in this Thesis.
- Chapter 3 defines the concepts of *CPU Capacity* and *CPU Accounting* for MT processors, identifying the main sources of inaccuracy in current CPU accounting mechanisms. Also, it introduces the *fair share of CPU capacity* which represents the hardware resources that a task should get access when the task runs alone onto a MT processor.
- Chapter 4 presents a novel CPU accounting mechanism, ITCA, for CMP processors that is aware of interference in shared hardware resources.
- Chapter 5 describes a novel CPU accounting mechanism for CMP+SMT processors, MIBTA, that extends the concepts of ITCA to SMT processors.
- Chapter 6 describes our concept of effective CPU capacity share for MT processor and proposes two OS CPU scheduling algorithms that are aware of the actual CPU capacity enjoyed by each scheduled task.

- Chapter 7 summarizes the main conclusions of this Thesis, presents our future work, and lists the relevant publications related to this Thesis.

Chapter 2

Platform, Tools, and Benchmarks

In this Chapter, we explain the tools and benchmarks we use to evaluate the current CPU accounting mechanisms and the CPU accounting mechanisms proposed in this Thesis.

2.1 Introduction

Researchers in this field propose and test novel techniques that could be included in the actual processor designs. In order to evaluate new techniques, computer architecture researchers usually use both simulation tools and benchmark suites. The simulation tools are used at different levels of detail, form circuit to system level, depending on the particular target system are studied. The benchmarks suites are representative of current and future applications that will be executed by the designed processor.

In this Thesis, several microarchitecture proposals are evaluated in term of both performance and accuracy for different workloads and processor configurations. For this reason, cycle-level microarchitecture simulators are used to model in detail the architecture of the processor and estimate the performance of a benchmark with different configurations. This simulator can be execution driven or trace driven. Execution-driven simulators usually have higher accuracy, but are time-consuming. Trace-driven simulators are less accurate, but are faster. As a result, we decided to choose a trace-driven simulator in order to get a trade-off between speed and accuracy in our experiments.

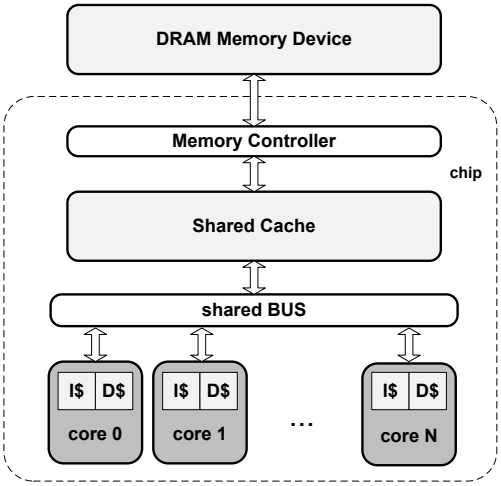


Figure 2.1: Modeled baseline microarchitecture

2.2 MPsim Simulator

To evaluate the performance of the different mechanisms shown in this Thesis and calculate the IPC, we use MPSim [1] (Multiple Purpose Simulator). This simulation tool includes a trace driven SMT simulator derived from SMTsim [64]. MPsim supports CMP processors and SMT processors and simulates the impact of executing along wrong paths on the branch predictor and the Instruction Cache. Also, MPsim models an in-order processor and an out-of-order processor.

In this Thesis, we model both CMP and CMP+SMT processors. Figure 2.1 shows our baseline CMP microarchitecture which consists of two or more cores which are connected to a shared L1-L2 bus, a shared last level cache, a shared memory controller, and a shared memory bandwidth. In our baseline CMP+SMT, each core supports SMT in which most on-core hardware resources are shared among all tasks running in the core. In both processor microarchitectures, the modelled pipeline each core is out-of-order as shown in Figure 2.2.

The fetch logic fetches instructions from the instruction cache in program order. Next, instructions are decoded and renamed in order to track data dependences. When an instruction is renamed, it is allocated an entry in the issue queues (Integer queue (IQ), Floating point queue (FPQ), and load/store queue (LSQ)) until all its operands are ready. Each instruction also allocates one entry in the Re-Order Buffer (ROB) and a physical register, if required. ROB entries are assigned in program order. When

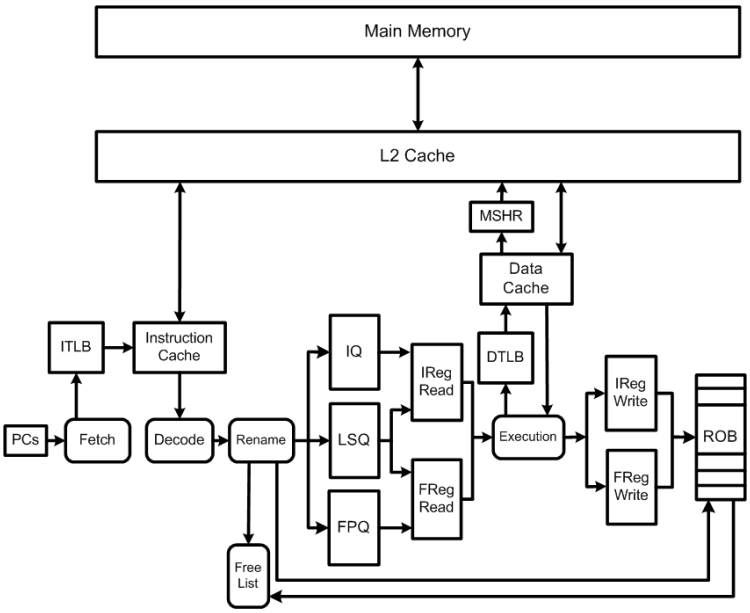


Figure 2.2: Block diagram of pipeline of a core

an instruction has all its operands ready, it is issued[1]: it reads its operands, executes, writes its results, and finally commits.

The data and the instruction caches are accessed with physical addresses. The data cache uses write back as write hit policy and write allocate as write miss policy. Caches are tagged with the identifier of tasks so that tasks do not share data and/or instructions.

The instruction fetch policy has an important role in SMT core because it determines which of the available tasks can fetch instructions each cycles. In our baseline SMT core, the modelled instruction fetch policy is ICOUNT [30] policy. The configuration of ICOUNT policy is $1.N$: up to N instructions can be fetched each cycle from a task.

The main shared resources in our baseline SMT core are the following. (1) The front-end bandwidth, which is assigned to tasks according to the instruction fetch policy. (2) The issue queues are shared between all tasks running on a core. (3) The issue bandwidth: we use is *first in first out* (oldest first) as issue policy. (4) The physical register files (Integer register file (IReg), and Floating point register file (FReg)) are

[1] In this Thesis, the term dispatch indicates the action of moving instructions to the issue queues. The term issue is applied to the action of submitting instructions from the issue queues to the backend of the processor.

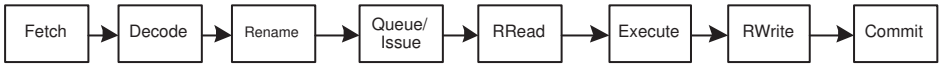


Figure 2.3: Pipeline stages in a core

common to all tasks. (5) Instruction and Data caches are shared between tasks. (6) Instruction and data Translation Lookaside Buffer (TLB) are also shared and tagged with task identifier, and (7) the ROB is shared among all tasks in a core.

The pipeline of our baseline core is composed of eight stages as shown in Figure 2.3. In our experiments, the decode stage takes up to four cycles and, in this way, the final number of stages in our pipeline is eleven.

2.3 Benchmarks

In the experiments performed during this research, we used the SPEC CPU 2000 benchmark suite [60] to evaluate our proposals in Chapter 4, and the SPEC CPU 2006 benchmark suite [61] in remaining Chapters. These benchmark suites are released by the Standard Performance Evaluation Corp. (SPEC), and are a worldwide standard for measuring and comparing computer performance across different hardware platforms.

SPEC CPU 2000 and SPEC CPU 2006 comprise two suites of benchmarks: SPEC CPU INT 2000 and SPEC CPU INT 2006 for compute-intensive integer performance and SPEC CPU FP 2000 and SPEC CPU FP 2006 for compute-intensive floating point performance. These benchmarks are selected from existing applications, representing high performance computing applications that stress the microarchitecture of the processor. Benchmark source codes run in different platforms so that performance comparisons can be made between different systems.

In our case, we run all benchmarks on an Alpha machine: an AlphaServer DS25 with two processors Alpha 21264C running at 1 GHz with the operating system Tru64 5.1b. Each program is compiled with the `-O2 -non_shared` options using DEC Alpha AXP-21264 C/C++ compiler and executed using the reference input set. Fortran programs are compiled with the DIGITAL Fortran 90/Fortran 77 compilers. We succeeded in compiling all the SPEC CPU 2000 benchmark suite and the SPEC CPU 2006 benchmark suite. We only had problems with the integer benchmark `483.xalancbmk` from SPEC CPU 2006 since its execution produced a known stack overflow problem.

In Tables 2.1, 2.2, 2.3 and 2.4, we give a short description of each benchmark in

Table 2.1: SPEC CPU INT 2000 benchmark description and simulation starting point (in millions of instructions) using the SimPoint methodology [54]

Benchmark	**Description**	**Input**	**Language**	**Fast forward**
164.gzip	Data compression utility	graphic	C	68.100
175.vpr	FPGA circuit placement and routing	place	C	2.100
176.gcc	C compiler	166.i	C	14.000
181.mcf	Minimum cost network flow solver	inp.in	C	43.500
186.crafty	Chess program	crafty.in	C	74.700
191.parser	Natural language processing	ref.in	C	83.100
252.eon	Ray tracing	cook	C++	57.600
253.perlbmk	Perl	splitmail.535	C	45.300
254.gap	Computational group theory	ref.in	C	79.800
255.vortex	Object Oriented Database	lendian1.raw	C	58.200
256.bzip2	Data compression utility	inp.program	C	13.500
300.twolf	Place and route simulator	ref	C	324.300

these suites together with the language in which the source codes were written. In the case of integer benchmarks, all the applications are written in C or C++, while in the case of floating point benchmarks, some of them are written in Fortran, C, C++, or a combination of C and Fortran codes. For example, in `435.gromacs` the only Fortran code is the inner loops (`innerf.f`) which typically account for more than 95% of the runtime.

Some benchmarks in the SPEC CPU 2006 suite are executed multiple times with different inputs for the reference test. We execute all benchmarks with the input sets suggested by Phansalkar et al. [46]. In this paper, the authors analyze all the reference input sets of each benchmark to find redundant input sets, and they choose the representative input set of the benchmarks that have multiple input sets. To obtain this representative input set, the authors find the input set that is the closest to the whole benchmark run (with all the input sets). In the SPEC CPU INT 2006 suite, the benchmarks with multiple input sets are `400.perlbench`, `401.bzip2`, `403.gcc`, `445.gobmk`, `456.hmmer`, `464.h264.ref` and `473.astar`. In the SPEC CPU FP 2006 suite, they are `416.games` and `450.soplex`. For each benchmark, we list the most representative input set in Tables 2.5 and 2.6 for the all the integer and floating point benchmarks, respectively.

Table 2.2: SPEC CPU INT 2006 benchmark description

Benchmark	Description	Language
400.perlbench	Devired from Perl V5.8.7	C
401.bzip2	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O	C
403.gcc	Based on gcc Version 3.2, generates code for Opteron	C
429.mcf	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport	C
445.gobmk	Plays the game of Go, a simply described but deeply complex game	C
456.hmmer	Protein sequence analysis using profile hidden Markov models	C
458.sjeng	A highly-ranked chess program that also plays several chess variants	C
462.libquantum	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm	C
464.h264ref	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2	C
471.omnetpp	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network	C++
473.astar	Pathfinding library for 2D maps, including the well known A* algorithm	C++
483.xalancbmk	Transforms XML documents to other docs using a modified Xalan-C++	C++

2.3.1 Simulation Time Reduction

The simulation of a whole benchmark takes a great amount of time which makes unaffordable to evaluate new techniques. To reduce simulation time, the most common approach is to select a smaller segment of every benchmark that is representative of the whole execution. Selecting these representative samples is an important issue [54, 67]. Random samples appear to be inadequate, while just choosing the beginning of a program could be incorrect due to initialization code. However, we know that a program execution consists of many different phases, where statistics such as cache or branch misses significantly change among them. Thus, representative segment (throughout this work we will call it *trace*) should represent major program phases. As a consequence, the traces allow reducing simulation time to analyse new techniques in this Thesis.

Another advantage of working with traces is that researches must only perform time consuming functional simulation of the whole application once, whereas they perform many times detailed timing simulations of the reduced trace. Therefore, the time of detailed simulation is significantly reduced.

Table 2.3: SPEC CPU FP 2000 benchmark description and simulation starting point (in millions of instructions) using the SimPoint methodology [54]

Benchmark	**Description**	**Input**	**Language**	**Fast forward**
168.wupwise	Quantum chromodynamics	wupwise.in	Fortran77	263.100
171.swim	Shallow water modeling	swim.in	Fortran77	47.100
172.mgrid	Multi-grid solver in 3D potential field	mgrid.in	Fortran77	187.800
173.applu	Parabolic/elliptic partial differential equations	applu.in	Fortran77	10.200
177.mesa	3D Graphics library	frames100 + msea.in	C	294.600
178.galgel	Fluid dynamics: analysis of oscillatory instability	galgel.in	Fortran90	175.800
179.art	Neural network simulation; adaptive resonance theory	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	C	13.200
183.equake	Finite element simulation; earthquake modeling	inp.in	C	27.000
187.facerec	Image processing	facerec.in	Fortran90	
188.ammp	Computer vision: recognizes faces	ammp.in	C	13.200
189.lucas	Computational chemistry	lucas2.in	Fortran90	30.000
191.fma3d	Finite element crash simulation	fma3d.in	Fortran90	10.500
200.sixtrack	Particle accelerator model	sixtrack.in	Fortran77	173.500
301.apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants	apsi.in	Fortran77	192.600

This idea drives the *SimPoint* methodology [54]. Sherwood et al. [54] explain how to detect a program's phases by using the Basic Block Vector (BBV) which counts how many times each basic block appears. Two phases are considered the same if Mannheim's distance between their BBVs is small. At the beginning, the execution of the program is split into a set of *intervals* of fixed size (10 million instructions). Using clustering algorithms, such as *random linear projection* or *k-means*, the samples are joined. The first algorithm is used to reduce the dimension of the BBV and, in that way, accelerate the k-means algorithm. This last algorithm is run for values of k between 1 and M (M is the maximum number of phases to use) and the intervals are grouped into phases. Using the Bayesian Information Criterion (BIC), which measures the goodness of fit of a clustering within a dataset, the smallest value of k with

Table 2.4: SPEC CPU FP 2006 benchmark description

Benchmark	Description	Language
410.bwaves	Computes 3D transonic transient laminar viscous flow	Fortran
416.gamess	Implements a wide range of quantum chemical computations. Test case does self-consistent field calculations using several methods	Fortran
433.milc	A gauge filed program: lattice gauge theory with dynamical quarks	C
434.zeusmp	a computational fluid dynamics code developed at NCSA for the simulation of astrophysical phenomena	Fortran
435.gromacs	Molecular dynamics: simulate Newtonian equations of motion for hundreds to millions of particles. The test case simulates protein Lysozyme in a solution	C, Fortran
436.cactusADM	Solves the Einstein evolution equations using a staggered-leapfrog numerical method	C, Fortran
437.leslie3d	Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme	Fortran
444.namd	Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I	C++
447.dealII	C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients	C++
450.soplex	Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models	C++
453.povray	Image rendering. The testcase is a 1280x1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function	C++
454.calculix	Finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library	C, Fortran
459.GemsFDTD	Solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method	Fortran
465.tonto	An open source quantum chemistry package, using an object-oriented design in Fortran 95. The test case places a constraint on a molecular Hartree-Fock wavefunction calculation to better match experimental X-ray diffraction data	Fortran
470.lbm	Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D	C
481.wrf	Weather modeling. The test case is from a 30km area over 2 days	C, Fortran
482.sphinx3	A widely-known speech recognition system from Carnegie Mellon University	C

a minimum BIC score is chosen. SimPoint chooses the representative of each phase that is closest to its centroid. Finally, these representatives are accurately simulated and the results are weighted by the size of each phase.

In the experiments performed in this Thesis, we made use of sampled simulation techniques in order to reduce simulation time without losing accuracy. In particu-

Table 2.5: The input sets for each benchmark in SPEC CPU INT 2006 and simulation starting point (in millions of instructions) using the SimPoint methodology [54]

Benchmark	Input	Fast Forward
400.perlbench	-I./lib checkspam.pl 2500 5 25 11 150 1 1 1 1	1439900
401.bzip2	input.program 280	107000
403.gcc	166.i -o 166.s	25500
429.mcf	inp.in	90700
445.gobmk	–quiet –mode gtp -i trevord.tst	50300
456.hmmer	–fixed 0 –mean 500 –num 500000 –sd 350 –seed 0 retro.hmm	14900
458.sjeng	ref.txt	822100
462.libquantum	1397 8	237000
464.h264ref	-d foreman_ref_encoder_main.cfg	382800
471.omnetpp	omnetpp.ini	683400
473.astar	rivers.cfg	220700
483.xalancbmk	–	–

Table 2.6: The input sets for each benchmark in SPEC CPU FP 2006 and simulation starting point (in Millions of instructions) using the SimPoint methodology [54]

Benchmark	Input	Fast Forward
410.bwaves	–	1668800
416.gamess	-i triazolium.config	2980700
433.milc	–	897600
434.zeusmp	–	17939
435.gromacs	-silent -deffnm gromacs -nice 0	588700
436.cactusADM	–	18497
437.leslie3d	-i leslie3d.in	637200
444.namd	–input namd.input –iterations 38 –output namd.out	1200
447.dealII	23	41900
450.soplex	-m3500 ref.mps	67400
453.povray	SPEC-benchmark-ref.ini	168600
454.calculix	-i hyperviscoplastic	1099500
459.GemsFDTD	–	31713
465.tonto	–	11500
470.lbm	3000 reference.dat 0 0 100_100_130_ldc.of	17900
481.wrf		2749700
482.sphinx3	ctlfile . args.an4	1740400

lar, we collected traces of the most representative 300 million instruction segment of each benchmark in SPEC CPU 2000 and 100 million instruction segment of each benchmark in SPEC CPU 2006, following the SimPoint methodology [54], as this methodology is widely accepted in the literature.

The fast forwards to apply to each benchmark in both SPEC CPU 2000 and SPEC CPU 2006 are shown in Table 2.1, 2.3, 2.5, 2.6, respectively. Finally, we are not able to create the traces from three benchmarks: `459.GermsFTDT`, `483.xalancbmk`, and `481.wrf` from SPEC CPU 2006.

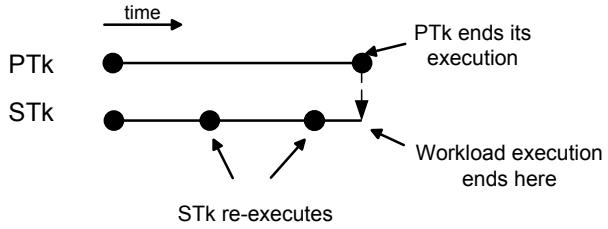


Figure 2.4: Execution mode of a workload

2.3.2 Workload Selection

In order to compute the accuracy of CPU accounting mechanisms, we generate workloads which consists of number of tasks. In each workload the first task in the tuple is the Principal Task (PTk) and the remaining tasks are considered as Secondary Task (STk). In every workload, we execute the PTk until completion. The other tasks are re-executed until the PTk completes as showed in Figure 2.4. This allows us to characterize the accuracy of proposals based on the type of the PTk and STks. It also allows to compare the execution time of the PTk when it runs in isolation with the time predicted by a CPU accounting mechanism once the workload simulation ends.

2.4 Comparison Metrics

The CPU accounting mechanisms presented in this Thesis are evaluated with two main metrics: *throughput* and *accuracy*. As the throughput metric, we measure weighted speedup [32], which is the sum of related IPC of each task in a workload. The related IPC of a task X is defined as $\frac{IPC_X^{MT}}{IPC_X^{isol}}$ where the IPC_X^{MT} is the IPC of the task X in a given workload in a multi-threaded processor, and the IPC_X^{isol} is the IPC of the task X when it runs in isolation in a system. The weighted speedup is calculated as shown in Equation 2.1

$$Weighted\ Speedup = \sum_{i=1}^{N} \frac{IPC_i^{MT}}{IPC_i^{isol}} \tag{2.1}$$

where N number of tasks in a given workload.

As the accuracy metric, we use the *Off estimation* which the relative error of the approximation time accounted to a task, TA, from a given CPU accounting mechanism, and the execution time of a task is running in isolation in a system, ET^{isol}. The off estimation is defined in Equation 2.2.

$$Off\ estimation = \left|1 - \frac{TA}{ET^{isol}}\right| \tag{2.2}$$

When a task is running in isolation in a system, ET^{isol} can be defined with two approaches that will be studied in Chapter 3: *full-share* and *even-share*. The former is the execution of a task that uses all hardware resources, ET^{isol}_{full}, and the latter is the execution time of a task that runs with an even part of the hardware resources, ET^{isol}_{even}.

Also, we report the average values of the five workloads with the worst off estimation, denoted *Avg5WOE*.

Chapter 3

Concept of CPU Capacity and CPU Accounting for Multi-Threaded Processors

In this Chapter, we discuss the definition of CPU capacity and CPU accounting for MT processors because these concepts are vague in these processors, whereas they are clearly defined in ST processors. To our knowledge, this Thesis represents the first effort in that direction.

3.1 Defining CPU Capacity

We define *CPU capacity* as the amount of 'CPU computing power' assigned to a task per time unit when the task is scheduled onto the CPU. In other words, the CPU capacity is the percentage of CPU assigned to a task running onto a processor.

We define *CPU capacity accounting* or simply *CPU accounting* to the process of measuring the CPU capacity assigned to a task. As shown in this section, CPU capacity can be easily defined for a ST architecture: in ST processors, a task gets access to 100 percent of the processor resources when it runs, since it has exclusive access to the processor. Under this scenario, CPU capacity can be measured simply using the CPU time of a task. CPU capacity is, however, more complex to define in MT processors, which is, in fact, the focus on this Chapter.

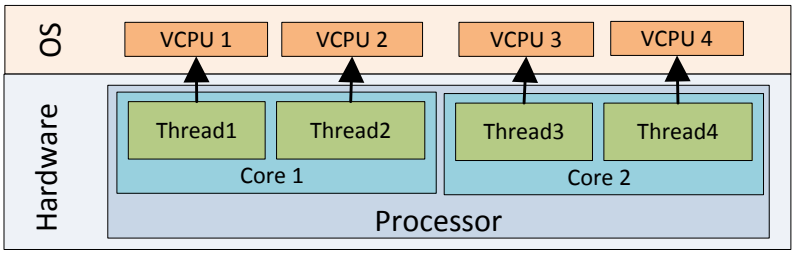


Figure 3.1: View point of a system for a MT processor

3.1.1 Abstracting CPU Capacity in MT processor

Many current OSes stick to the SMP model for MT processors [7]. In this model, the OSes equally perceive each core in a CMP or each hardware thread in an SMT as a virtual CPU (VCPU), which is assumed to deliver the same performance as a real processor (i.e. it suffers no slowdown when tasks run onto other VPCUs), and do not have any bias toward one VCPU with respect to others as illustrated in Figure 3.1. However, this model assumes that all processor resources are available for a task running onto a VCPU without considering the interference between tasks caused by shared hardware resources. This assumption is not accurate, because VCPUs from a MT processor share processor resources. Therefore, CPU capacity is shared among VCPU from a processor. This breaks the principle of accounting in MT processors as shown in Section 1.1.

Figure 3.2 shows a synthetic example with three tasks running on a system with a two-core processor. Tasks A, C, and D have the same priority. Task A, which is assigned a core (VCPU 0), runs concurrently with task C and D that are assigned to a different core (VCPU 1). The processor resources are shared between tasks running simultaneously onto the processor. We note that a task can receive different CPU capacity depending upon the hardware requirements of other tasks running onto the processor. For example, task A gets more CPU capacity when running with task C than with task D. This may be because task D needs more hardware resources than task C.

The main problem to provide the principle of accounting in MT processors is that the execution time of tasks is highly influenced by the on-chip shared resources, which tasks *compete for*. As a result, the execution time of a task, and hence, the CPU capacity assigned to the task does not only depend upon the time it runs onto the processor, as it is the case in ST uniprocessor systems, but also on the other tasks it runs with (the workload). The workload in which a task runs determines the interference

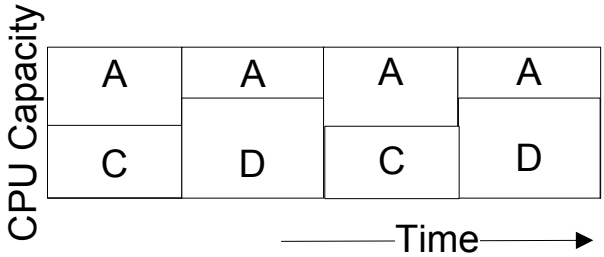


Figure 3.2: Synthetic example with three tasks that are scheduling onto two VCPUs and receive different CPU capacities

when accessing shared resources it suffers. Moreover, the non-linear relation between the resources a task receives and the progress it does, makes the computation of CPU capacity hard [33, 13, 16].

3.1.2 CPU Capacity

Finding an appropriate definition of *CPU capacity* is challenging, since (1) CPU capacity has not been defined in the literature (it is an abstract concept), and (2) it is hard to estimate the CPU capacity a task receives as it shares the CPU with other tasks. Next, we provide two novel definitions of CPU capacity, and also provide a qualitative evaluation of each CPU capacity definition.

CPU capacity as time. This approach is an extension of ST processor systems. It assumes that all the CPU capacity is available when a task is running onto a VCPU. As a consequence, the approach makes use of time as a proxy to measure CPU capacity. As we have shown, this definition is fundamentally flawed in MT processors.

CPU capacity as resource utilization. Intuitively, CPU capacity can be understood as resource utilization, meaning that tasks with the same priority should be given access to the same percentage of shared resources. Note that this approach does not imply physically partitioning the shared resources to ensure that each task receives the assigned amount of resources. This could be done, but it is not mandatory. As an alternative, tasks could be allowed to run with no resource utilization constraints, but with the proper hardware support to track the actual resource utilization (occupancy).

This interpretation of CPU capacity requires hardware support to track the resource utilization per task and requires adequate control policies for these resources. Let's assume a multi-core processor with N cores, in which each core has its private first level instruction and data cache, with a shared second level cache. The CPU capacity a task i receives is a function of the utilization of the shared resources (e.g.

cache and memory bandwidth):

$$CPU\_Capacity_i = f\left(RU_i^{cache}, RU_i^{memBW}\right) \quad (3.1)$$

Deriving a formula that computes overall CPU resource utilization based on the per-resource utilization is complex. This is specially the case in processors with high degree of resource sharing such as SMT processors. Moreover, some tasks are more sensitive to sharing a given resource than others, so two tasks receiving the same percentage of each shared resource may make different forward progress.

It is also the case that current processors do not provide such per-task resource utilization support. So, specific hardware support is required for that purpose. The overhead of that hardware is high when shared resources are accessed on a cycle-basis, for instance the core resources in an SMT.

CPU capacity as the CPU progress. The limitations of the previous definition of CPU capacity motivates us to propose an alternative definition. In this case, CPU capacity is mapped into *task execution progress that the task would do with a 'fair share' of resources*.

Let's assume that a task X runs for a period of time in a MT processor, TR_{X,I_X}^{MT}, in which it executes I_X instructions. The relative fair progress that task X has in this interval of time (P_{X,I_X}^{MT}) is expressed as

$$P_{X,I_X}^{MT} = \frac{TR_{X,I_X}^{fair-share}}{TR_{X,I_X}^{MT}} \quad (3.2)$$

The relative progress can also be expressed with the equation

$$P_{X,I_X}^{MT} = \frac{IPC_{X,I_X}^{MT}}{IPC_{X,I_X}^{fair-share}} \quad (3.3)$$

in which IPC_{X,I_X}^{MT} and $IPC_{X,I_X}^{fair-share}$ are the IPC of the task X when executing the same I_X instructions in the MT processor (likely shared with other tasks) and with a fair share of CPU capacity, respectively.

3.1.3 Fair Share of CPU Capacity

Several definitions of the actual fair share of CPU capacity are possible. In this Thesis, we have advocated for the following:

- **Full-Share Approach**. Under each CPU capacity definition, this definition of

fair share incarnates as follows

  - Taking CPU progress as CPU capacity, a task is considered to progress adequately (in a fair manner) when its execution progress is the same as it would be when the task runs in isolation onto the CPU. This accounting approach establishes that a task can access all processor hardware resources when it runs onto the CPU.
  - Taking resource utilization as CPU capacity. Under this definition of fair share of CPU capacity, a task should have access to all processor resources during its execution. That is, a task accesses a fair share of the CPU capacity when it has access to all processor resources.

- **Even-Share Approach**: Under each CPU capacity definition, this definition of fair share incarnates as follows

  - Taking CPU progress as CPU capacity, the even-share approach considers that the progress that a task should do is equivalent to the progress that the task would do with an even share of the processor resources.
  - Taking resource utilization as CPU capacity, a fair share is mapped into an even part of shared resources. That is, for each shared resources r_i, the CPU capacity a task should receive would be $1/N_i$ of the resource capacity and bandwidth, where N_i is the number of tasks that can simultaneously access r_i.

3.1.3.1 Putting It All Together

When using CPU capacity based on progress with full-share approach (denoted *full-share accounting*) or with even-share approach (denoted *even-share accounting*), the main challenge is determining dynamically, while a task X is simultaneously running with other tasks onto a processor, the speed (or IPC) it would have taken X to execute the same instructions if it had run with a fair share of the resources. This challenge is studied in Chapters 4 and 5. In Chapter 4, we provide novel CPU accounting hardware mechanisms, one for full-share accounting and one for even-share accounting respectively. In Chapter 5, we provides a novel CPU accounting mechanism for full-share accounting.

When using CPU capacity based on resource utilization with both definitions of fair share, to the best of our knowledge, no hardware support has been provided to

measure the resource utilization of tasks with the required level of detail (per cycle information), and no methods exist to derive the resource utilization of running tasks for chip resources.

3.2 Defining CPU Accounting

The CPU accounting of a task measures the CPU capacity utilized by the task during its execution in a MT processor. The CPU accounting serves as a bridge between the hardware and software, for instance the OS which works with the notion of time. So under our reference (i.e. full-share and even-share accounting) the CPU accounting mechanism measures the time that a task would need to execute the same instructions that have been executed onto MT processor in a given period when running in isolation with access to either all processor resources or even part of the processor resources.

The CPU accounting of a task X, TA^{MT}_{X,I_X}, is the time it would take this task to execute in isolation I_X instructions (either with all processor resources or an even part of the processor resources), denoted TR^{isol}_{X,I_X}, that have been run onto MT processor, TR^{MT}_{X,I_X}, and the relative progress of task X is P^{MT}_{X,I_X}. The CPU accounting is expressed as

$$TA^{MT}_{X,I_X} = TR^{MT}_{X,I_X} \cdot P^{MT}_{X,I_X} \tag{3.4}$$

from which we conclude

$$TA^{MT}_{X,I_X} = TR^{ISOL}_{X,I_X} \tag{3.5}$$

This makes the CPU accounting of a task independent from the rest of the workload, regaining the principle of accounting for MT processors.

For instance, if a task A executes 5 milliseconds in a MT processor and the CPU accounting measures 4.5 milliseconds, it means that the task would run for 4.5 milliseconds if it run in isolation on the processor. The task progressed 90 percent of the time, and in the remaining time, the task was stalled due to interferences with other tasks. In contrast, in a ST processor, the task A would have progressed 100 percent of the time as it would run alone in the processor.

Under full-share accounting, a CPU accounting mechanism should take into account the interference due to activities of other tasks in order to obtain the CPU accounting of tasks accurately. For instance, assuming that for a task A, its working set fits in the LLC when running in isolation, and for other task B its working set does not fit in the LLC. If both tasks run onto a two-core CMP processor with shared LLC, the CPU capacity of task A will be different from in isolation because the task B mono-

polizes the LLC. However, the CPU accounting of the task A should be similar to or the same as in isolation.

Under even-share accounting, CPU accounting mechanisms should measure the CPU accounting of a task in its even-share of resources without split hardware resource physically, but logically. For example, in a CMP processor, a task can progress more than in isolation (even-share of resources) because the task can utilize more hardware resources than its even-share of resources. This scenario should be taken into account by CPU accounting mechanisms.

In conclusion, both accounting approaches can be utilized as reference in order to measure the CPU accounting of tasks in MT processors. Choosing one of the approaches depends upon the objectives of CPU accounting mechanisms and the systems.

3.3 Summary

This Chapter has introduced the concept of CPU capacity in MT processors as well as suitable definitions of CPU capacity and of fair share of CPU capacity for MT processors. Moreover, we have introduced the concept of CPU accounting, which measures the CPU capacity of a task that is scheduled onto a MT processor as if the task is alone running onto the processor.

Chapter 4

CPU Accounting for Multi-Core Processors

In this Chapter, we firstly analyse the classical approach for CMP processors, and propose a novel CPU accounting hardware mechanism, *InterTask Conflict-Aware* (ITCA) accounting, which improves the accuracy in measuring CPU capacity of the Classical Approach (CA) for CMPs. On a modelled 2-, 4-, and 8-core CMP, ITCA reduces the inaccuracy of the classical approach from 7.0%, 13%, and 16%, to 2.4%, 3.7%, and 2.8%, respectively. Second, we evaluate the accuracy of ITCA in conjunction with dynamic cache partitioning algorithms [49]. Third, we show that we can implement an improvement of ITCA with small changes, denoted I^2TCA, and that I^2TCA reduces the average off estimation of ITCA. Finally, we show that under even-share accounting, I^2TCA works well with small changes.

4.1 Introduction

Chip MultiProcessors (CMP) or multi-core processors have emerged as the dominant architecture choice for modern computing systems. The benefits of multi-core processors are high power efficiency and less design costs, due to multiplex simpler cores in a processor (chip). In addition, cores have less power-hungry hardware structures, and as a result, average power consumption is reasonable.

Due to gap between the processor and the memory speed, the performance of a task may be degraded. In order to face this scenario, the cache hierarchy is included in most currently processor. The cache hierarchy store data recently used by a task in order take advantage of the temporal and spatial locality of the task.

In CMP processors, the cache hierarchy is normally organized in a first level of in-

struction and data cache private to each core. The last level cache is generally shared among different cores in the processor. Shared caches increase resource utilization and system performance. Large caches improve performance and efficiency by increasing the probability that each task can access data from closer level of the cache hierarchy. Moreover, they allow a task to make use of the entire cache.

Since tasks share the same address space in LLC, they compete for space in the LLC. As a consequence, the contention in the LLC may degrade tasks' performance and may affect CPU accounting as shown in Section 1.1. In current CPU accounting mechanisms, the contention in LLC does not take into account CPU accounting and hence, their accuracy is poor. In this Chapter, we present a new CPU accounting mechanism, named ITCA (Intertask Conflict-Aware) and ITCA maintains the principle of accounting in CMP processors.

ITCA [33] has been developed based on our understanding of the processor architecture. In this Chapter, we show that our intuition in developing ITCA is correct by making a complete design space exploration of the possible combinations of *Hardware Resource Status Indicator* (HRSI) states. This exploration confirms that ITCA provides reasonable accurate results. It also shows that we can implement an improvement of ITCA with small changes, denoted I^2TCA, and that I^2TCA reduces the average off estimation of ITCA mainly in the five worst workloads: from 32% to 13% in the 2-core configuration, from 35% and 17% in the 4-core configuration and from 20% to 14% in the 8-core configuration. Furthermore, I^2TCA can work well in even-share accounting in order to maintain the principle of accounting.

The rest of the Chapter is organized as follows. Section 4.2 provides a comprehensive analysis of the CPU accounting accuracy of the CA. Section 4.3 describes our proposal of CPU accounting with improved accuracy in computing the CPU capacity in CMP processors. The experimental methodology and the results of our simulations are presented in Section 4.4. Section 4.5 evaluates the effects of considering different HRSIs in the CPU accounting done by ITCA. Next, we evaluate the accuracy of the CA and ITCA under even-share accounting to compute CPU capacity in Section 4.6. Section 4.7 studies other issues regarding CPU accounting such as performance counters and accounting for multi-threaded tasks. Finally, Section 4.8 concludes the study.

4.2 Background

In this section, we study the accuracy of the CA for CMP processors. To the best of our knowledge, the CA is the only accounting mechanism for CMP processors for

open source OSes such as Linux.

4.2.1 The Classical Approach

The classical approach accounts tasks based on the time they run onto a CPU, instead of the progress each task performs. Therefore, the CA implicitly assumes that running tasks have full access to the processor resources. However, each task shares resources with other tasks when running in a MT processor, which leads to intertask conflicts[1]. As a consequence, a task takes longer to finish its execution than when it runs in isolation, resulting in longer accounting time. For this reason for a task X, the CA leads to *over-estimation*

$$TA^{CA}_{X,I_X} = TR^{CMP}_{X,I_X} > TR^{ISOL}_{X,I_X} \tag{4.1}$$

A task has no over-estimation only if it executes with no slowdown in the CMP processor with respect to its execution in isolation, in which case

$$TA^{CA}_{X,I_X} = TR^{CMP}_{X,I_X} = TR^{ISOL}_{X,I_X} \tag{4.2}$$

In order to illustrate the concept of over-estimation and without loss of generality, we assume a dual-core in-order processor. The two cores share the L2 cache, whereas the first level data and instruction caches are private to each core. We further assume an L2 miss latency of ten cycles and an L2 hit latency of one cycle. Even if these latencies are not representative of any current processor, they are perfectly valid for the purpose of illustrating the problem of CPU accounting. Finally, we assume that the execution time of a task X when running in isolation (TR^{ISOL}_{X,I_X}) is known. In remaining Chapter, all these assumptions are removed.

In Figure 4.1, each square represents a processor cycle. The *progress* row shows whether a task progresses in each cycle or not. If the task executes any instruction in that cycle, it is marked as one. Otherwise, it is marked as zero. The values in the CA row show the CPU time accounted to each task. Figure 4.1 (a) shows the situation in which a task X runs in isolation and executes a memory access that hits in the L2 cache. Under this scenario, the memory access resolves in one cycle, so X is

[1]Throughout this Thesis, we refer to *intertask* resource conflicts to those resource conflicts that a task suffers due to the interference of the other tasks running at the same time. For example, a given task X suffers an intertask L2 cache miss when it accesses a line that was evicted by another task, but would have been in cache, had X run in isolation. Likewise, *intratask* resource conflicts denote those resource conflicts that a task suffers even if it runs in isolation. These are conflicts inherent to the task.

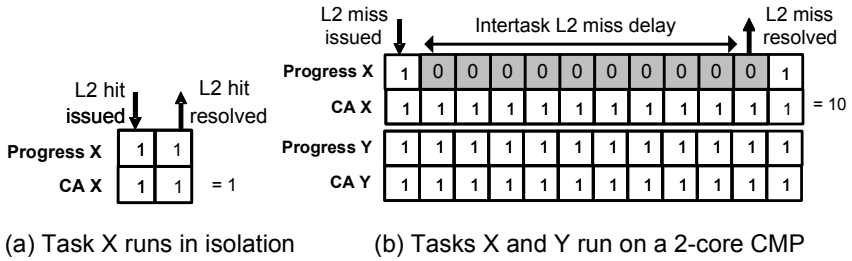


Figure 4.1: Synthetic example to illustrate over-estimation with the CA. The example highlights the effect of an intertask L2 miss on a 2-core CMP processor

accounted for one cycle for processing the memory access.

Figure 4.1 (b) shows another situation in which X runs in one core and a task Y runs in a second core. In this case, we assume that task Y evicts the data belonging to X from the L2 cache, causing the previous L2 hit of X to become an intertask L2 miss. This intertask miss causes X to stall its execution (dark squares) until it is resolved ten cycles later. Under this scenario, X takes longer to serve the memory access and is accounted for ten cycles. In this particular example, the intertask resource conflict causes an over-estimation of the accounted time to task X. In this example, it is assumed that task Y does not suffer any intertask miss, doing the same progress as in isolation.

The main source of over-estimation in on-chip CMP processors is the delay caused by intertask conflicts and, in particular, intertask L2 misses.

In order to show this phenomenon, we derive an empirical relationship between over-estimation in CPU capacity and intertask L2 misses in the CA in CMP processors. We run all possible 2-task workloads from SPEC CPU 2000 benchmarks. For each workload, we compute the CPU capacity for PTk according to the CA, that is the execution time in CMP. For each task, we obtain the average percentage of stalled cycles due to intratrask and intertask L2 misses. Next, we sort the tasks in decreasing order according to the off estimation introduced by the CA, as shown in Figure 4.2. We observe that the 9 tasks with more than 5% stalled cycles due to intertask L2 misses are the 9 tasks with the highest off estimation. Benchmarks that suffer more intratask misses can overlap these misses with intertask misses, consequently suffering less off estimation. This is the case for *art*. Finally, the 9 tasks with less than 1% stalled cycles due to intertask L2 misses present less than 1.1% off estimation. Overall, we observe a clear influence of intertask L2 misses on the accuracy of the CA.

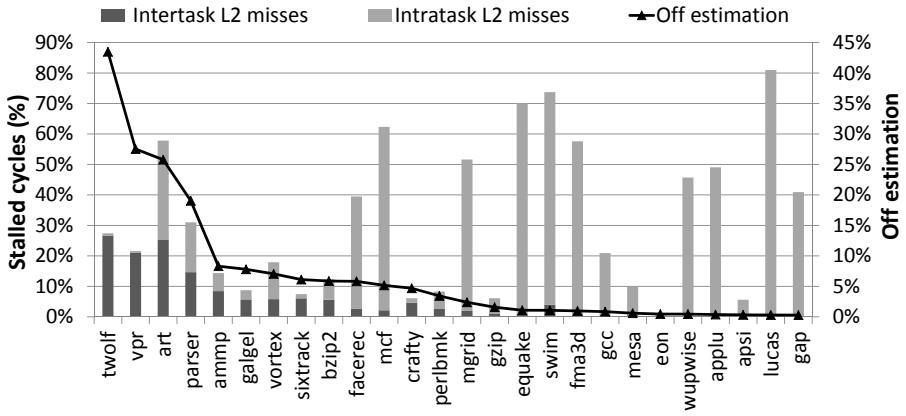


Figure 4.2: Correlation between over-estimation and stalled cycles due to intratask and intertask L2 misses in the CA in a CMP processor

4.3 InterTask Conflict-Aware Accounting

The target of our proposal, *InterTask Conflict-Aware* (ITCA) accounting, is to accurately estimate the CPU capacity accounted to a task in CMPs. The basic idea of ITCA is to account to a task only those cycles in which the task is not stalled due to an intertask L2 cache miss. In other words, a task is accounted CPU cycles when it is progressing or when it is stalled due to an intratask L2 miss. The next paragraphs provide a detailed discussion of when the accounting of a task is stopped and resumed.

L2 data misses: We consider a task is in one of the following states: (s1) It has no L2 data (cache) misses or it has only intratask L2 data misses in flight; (s2) It has only intertask L2 misses in flight; and (s3) It has both intertask and intratask L2 misses in flight simultaneously.

In the state (s1) we do normal accounting because there is no intertask L2 miss in flight. We consider a task is not progressing, and hence, it should not be accounted in state (s2). This means that accounting is stopped when the task experiences an intertask L2 miss and it cannot overlap its stall with any other intratask L2 miss. We resume accounting for the task when the intertask L2 miss is resolved or the task experiences an intratask L2 miss, in which case the task overlaps the memory latency of the intertask L2 miss with at least one intratask L2 miss.

In state (s3), intertask L2 misses overlap with intratask L2 misses. As a consequence, in general we do a normal accounting to the task in that state. However, when an intertask L2 miss becomes the oldest instruction in the ROB and the register renaming stage is stalled, the task loses an opportunity to extract more Memory Level

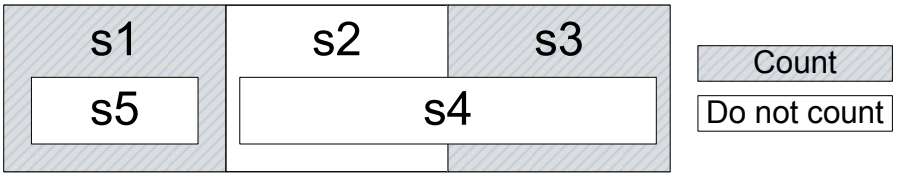


Figure 4.3: Accounting decision for all possible states

Parallelism (MLP). For instance, let us assume that there are S instructions between the intertask L2 miss in the top of the ROB and the next intratask L2 miss in the ROB. In this situation, if the task had not experienced the intertask L2 miss it would have executed the S instructions after the last instruction currently in the ROB. Any L2 miss in those S instructions could have been sent to memory, increasing the MLP. We take care of this lost opportunity of extracting MLP by stopping the accounting of a task if the instruction in the top of the ROB is an intertask L2 miss and the register renaming stage is stalled. We call this condition state (s4). It can be the case that such stalls end up having no impact in the accounting because those instructions could not have been executed anyway if an intratask miss is oldest instruction in the ROB. In this case, it is included in state (s3), and hence, we do normal accounting.

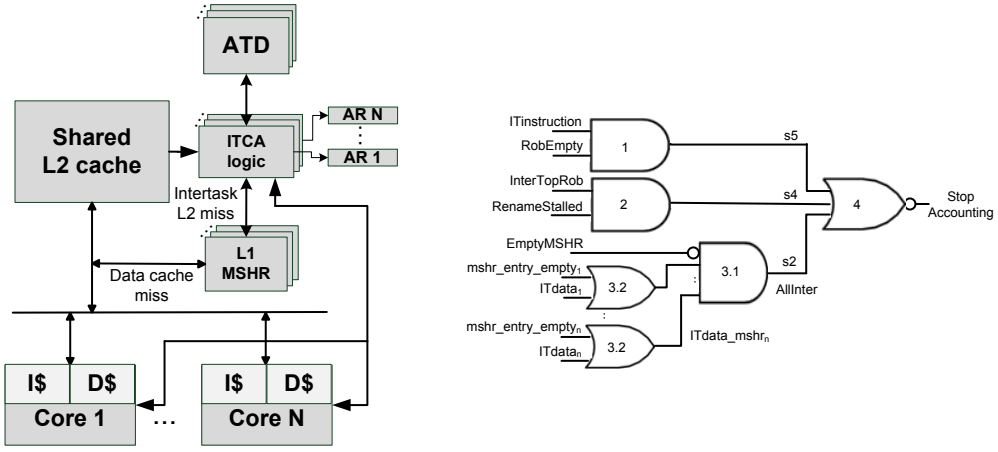
L2 instruction misses: ITCA also stops accounting to a task when the ROB is empty because of an intertask L2 instruction cache miss (s5). In our processor setup instruction cache misses do not overlap with other instruction cache misses. That is, at every instant, there is at most one in flight instruction miss per task. Hence, on an intertask L2 instruction miss we consider that the task is not progressing because of an intertask conflict, and hence, we stop its accounting.

In short, the accounting decision in the defined five states is illustrated in Figure 4.3. Note that state (s5) is part of state (s1), since when the ROB is empty, no L2 data cache miss can be in flight. Finally, state (s4) can only occur when there are some intertask L2 data misses in flight and, consequently, is contained in states (s2) and (s3).

4.3.1 Hardware Implementation

Figure 4.4 shows a sketch of the hardware implementation of our proposal, which makes use of several HRSIs. Next, we explain in depth the different parts of our approach.

Detecting intertask misses: We keep an *Auxiliary Tag Directory* (ATD) [49] for each core (see Figure 4.4 (a)). The ATD has the same associativity and size as the



(a) Baseline processor architecture (b) Logic to stop accounting

Figure 4.4: Hardware required for ITCA

tag directory of the shared L2 cache and uses the same replacement policy. It stores the behaviour of memory accesses per task in isolation. While the tag directory of the L2 cache is accessed by all tasks, the ATD of a given task is only accessed by the memory operations of that particular task. If the task misses in the L2 cache and hits in its ATD, we know that this memory access would have hit in cache if the task had run in isolation [37]. Thus, it is identified as an intertask L2 miss.

Tracking intertask misses: We add one bit called $ITdata_i$ bit in each entry i of the Miss Status Hold Register (MSHR). The ITdata bit is set to one when we detect an intertask data miss. Each entry of the MSHR keeps track of an in flight memory access from the moment it misses in the data L1 cache until it is resolved.

On a data L1 cache miss, we access the L2 tag directory and the ATD of the task in parallel. If we have a hit in the ATD and a miss in the L2 tag directory, we know that this is an intertask L2 cache miss. Then, the ITdata bit of the corresponding entry in the MSHR is set to 1. Once the memory access is resolved, we free its entry in the MSHR.

When the ROB is empty due to an intertask L2 cache instruction miss, we stop accounting cycles to this task. For that purpose, we use a bit called *ITinstruction* that indicates whether the task has an intertask L2 cache instruction miss or not.

4.3.2 CPU Accounting in ITCA

We stop the accounting of a given task when:

1 The ROB is empty because of an intertask L2 instruction cache miss (gate (1) in Figure 4.4 (b) that implements condition (s5)). *RobEmpty* is a signal that is already present in most processor architectures, while *ITinstruction* indicates whether or not a task has an intertask L2 cache instruction miss.

2 The oldest instruction in the ROB is an intertask L2 data cache miss and we have a stall in the register renaming stage (gate (2) in Figure 4.4 (b) that implements condition (s4)). The HRSI denoted *InterTopRob* tracks the first condition, while *RenameStalled* monitors the second one. Storing a bit to track intertask L2 misses might require one bit per ROB entry.

3 All the occupied MSHR entries belong to intertask L2 data misses (gates (3.1) and (3.2) in Figure 4.4 (b) implement condition (s2)). To determine this condition, we check whether every entry i of the MSHR is not empty ($mshr\_entry\_empty_i = 0$) and contains an intertask L2 data miss ($ITdata_i = 1$). By making an AND operation of $ITdata\_mshr_i$ and a signal showing whether the entire MSHR is empty (tracked with the HRSI *EmptyMSHR*), we determine if we have to stop the accounting for the task.

Finally, if any of the gates (1), (2) or (3.1) returns 1, we stop the accounting. Otherwise, we account the cycle normally to the task as occurs in states (s1) and (s3).

In a 2-core CMP, ITCA accounts for every spent cycle in three possible ways: (1) Each task is accounted for the cycle when both tasks progress (the cycle is accounted twice, one for each task). (2) Only one task is progressing and the cycle is accounted only to it. (3) The cycle is not accounted to any task when none of them is progressing.

The cycles accounted to each task in each core are saved into a special purpose register per core, denoted *Accounting Register (AR)* (see Figure 4.4 (a)), which can be communicated to the OS. This register is a read-only register and can be managed by the OS similarly to the Time Stamp Register in Intel architectures. From the OS point of view working with ITCA is similar to working with the CA. On every task switch, the OS reads the Accounting Register of each $task_i$ (AR_i), where AR_i reports the time to account this task. With this information, the OS updates system's metrics. The OS can alternatively be changed to use the information provided by ITCA similar to [16]. On a task migration, both the ATD and the cache require some time to warm up but we expect this overhead to be low.

Table 4.1: Simulator baseline configuration

Number of cores	2, 4 & 8
Fetch bandwidth	8 inst. per cycles
Issue queues sizes	64 int, 64 fp, 64 ld/st
Execution units	6 int, 3 fp, 4 ld/st
Back end	196 int/fp phys. registers, 512-entry ROB
Branch predictor	Perceptron 256 global-entry, 40 global-H, 4K local-entry, 14 local-H, 100-entry RAS
Target frequency	2.0GHz
Icache (per core)	64KB, 2 ways, 1 bank, 128B line, 1-cycle access
Dcache (per core)	32KB, 4 ways, 1 bank, 128B line, 1-cycle access
L2 cache (Shared)	2MB, 4MB, and 8MB, 16 ways, 4 banks, 128B line, 15-cycle access
MSHR	32 entries
Memory latency/Bandwidth	300-cycle access, 12.1 GB/s

4.4 Evaluation Results

In this section, we show the accuracy results of CPU accounting mechanisms: CA and ITCA, the hardware overhead for ITCA, the accuracy of ITCA in conjunction with dynamic cache partitioning algorithms, and study the memory bandwidth sensitivity for ITCA.

In order to evaluate different sections, we use three processor setups are shown in Table 4.1: a 2-core CMP with a 2MB L2 cache, a 4-core CMP with a 4MB L2 cache, and an 8-core CMP with an 8MB L2 cache.

We feed our simulator with traces collected from the whole SPEC CPU 2000 benchmark suite (see Chapter 2 for more details). From these benchmarks, we generate 2-task, 4-task and 8-task workloads. We classify benchmarks into two groups depending upon their cache behaviour. Benchmarks in the memory group (denoted M) are those presenting a high L2 cache miss rate in isolation ($MPKI_{ISOL} > 1$), while benchmarks in the ILP group (denoted I) have low L2 cache miss rate as shown in Table 4.2. From these two groups, we generate different workload types denoted V_W, where V is the type of the PTk and W is the type of the STk. We distinguish three combinations of STk: ILP, MEM and MIX. ILP combinations contain only ILP benchmarks, MEM combinations contain only memory-bound benchmarks and MIX combinations contain a mixture of both. For example, in the group M_ILP with 4 tasks, the PTk is memory bound and the 3 STks are ILP bound. Note that for the 2-core configuration there is only one STk and, consequently, we can only evaluate STks belonging to groups ILP or MEM. In total, we use 576 2-task workloads, 192 4-task workloads and 96 8-task workloads, randomly generated.

Table 4.2: Benchmarks' cache behaviour (2MB L2 cache)

(a) Memory-group benchmarks

Type	Benchmark	MPKI
INTEGER	mcf	85.64
	gzip	1.81
	gap	1.01
FP	fma3d	73.93
	swim	14.73
	equake	10.40
	lucas	10.10
	art	7.39
	applu	6.13
	mgrid	2.99
	facerec	2.64
	wupwise	1.26

(b) ILP-group benchmarks

Type	Benchmark	MPKI
INTEGER	parser	0.56
	gcc	0.55
	vortex	0.43
	perlbmk	0.14
	bzip2	0.08
	twolf	0.04
	vpr	0.04
	crafty	0.04
	eon	0.00
FP	apsi	0.54
	ammp	0.50
	galgel	0.24
	mesa	0.20
	sixtrack	0.04

4.4.1 Accuracy Results

Figure 4.5 shows the off estimation of ITCA and the CA for our 3 processor setups. We show the average results of each group. The bars labelled AVG represent the average of each CMP configuration for all the groups. While on average the CA has an off estimation of 7.0% (2 cores), 13% (4 cores) and 16% (8 cores), ITCA reduces it to less then 2.4% (2 cores), 3.7% (4 cores) and 2.8% (8 cores). These results indicate that ITCA provides a good measure of the progress each task makes with respect to its execution in isolation, since ITCA takes into account intertask L2 misses. Moreover, ITCA reduces the inaccuracy in the worst five cases: the Avg5WOE metric is 117% (2 cores), 91% (4 cores) and 94% (8 cores) for the CA and only 32% (2 cores), 35% (4 cores) and 20% (8 cores) for ITCA.

Next, we observe that the accuracy of the CA is worse when the PTk is in the ILP group and all the STks are in the MEM group (group I_MEM). This is due to the fact that some of the ILP tasks experience a lot of hits in the L2 cache when they run in isolation. When they run simultaneously with MEM tasks, which make an intensive use of the L2 cache, the ILP tasks suffer a lot of intertask misses. As a consequence, the ILP task suffers an increase in its execution time, which affects the accuracy of the CA. When the PTk is in the MEM group, it already suffers a lot of L2 misses in isolation, so that the increase in the number of L2 misses when it runs with other MEM tasks is relatively lower.

Next, we observe that the inaccuracy of the CA for a given group increases with

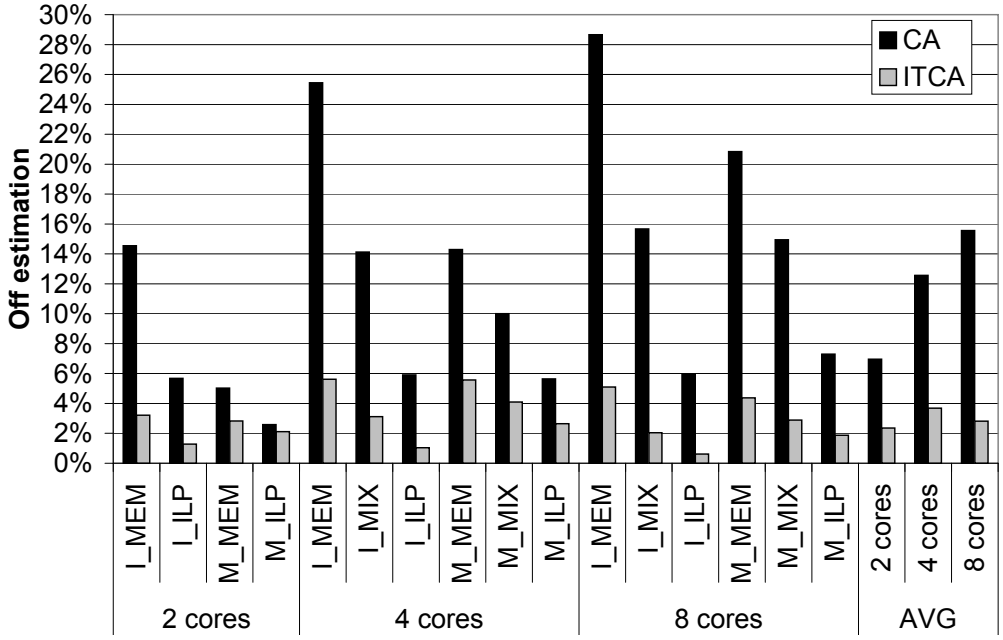


Figure 4.5: Off estimation of the CA and ITCA for 2-, 4- and 8-core CMPs with a shared 2MB, 4MB and 8MB L2 cache, respectively

the number of cores. Even if in our processor setups for 2, 4, and 8 cores the average cache space per task is kept the same (1MB per core), the average off estimation of the CA increases from 7.0% (2 cores) to 16% (8 cores). The main reason for that behaviour is that having more tasks sharing the cache increases the probability that one of them thrashes the other tasks, which will lead to higher off estimations in the CA. The capacity of the L2 cache is not enough to store all the data of the tasks running simultaneously and for example, the off estimation of the group I_MEM in 2 cores is 15%, but reaches 29% in 8 cores.

4.4.2 Reducing the ATD's Overhead

The overhead of our baseline ATD (Auxiliary Tag Directory) is 30KB per core (15-bit tag, 1024 sets, 16 ways per set). This is still a substantial area in a chip. In order to reduce the area requirements, we implement two simplified versions of the ATD.

First, we save only a subset of the address' tag bits of each memory access in each entry of the ATD. On an access to the L2 cache, we only compare this subset of bits of the tag between the ATD and the L2 directory. This scheme introduces false hits when the subset of bits compared are equal in the ATD and in the L2 tag directory, but

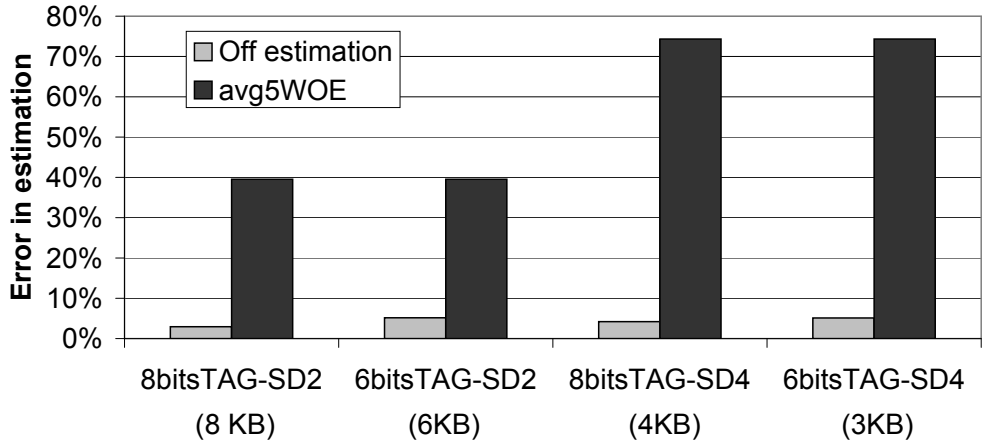


Figure 4.6: Effect of reducing ATD overhead on accuracy

the other bits of the tag are not. As a consequence, this scheme confuses some actual intertask L2 misses with intratask L2 misses, and vice versa.

Second, we use a sampled version of the ATD, denoted sATD [49]. This scheme monitors only a subset of the cache sets (sampled sets), providing similar results to the ATD in terms of performance [49]. When using sATDs with ITCA, for accesses to non-sampled sets we cannot determine whether they are intertask or intratask misses. In this situation, ITCA does not consider these misses in the accounting task. In Chapter 5, we will study a new version of sATD that consider this situation.

Figure 4.6 shows the area reduction and accuracy degradation of the simplified versions of the ATD, with respect to our baseline ATD. We use addresses of 32 bits, so the tags have 15 bits in the L2 cache. A good trade-off is when we sample every 2 sets and the ATD has tags of 6 bits (denoted 6bitsTag-SD2). In this case, we reduce the size of the ATD to 6KB, and increase the average off estimation and Avg5WOE to 5.2% and 40%, respectively. Recall that in this configuration, the CA leads to an average off estimation and Avg5WOE of 7.0% and 117%, respectively. Depending upon the hardware budget available, different trade-offs are possible. For example, if 8KB of area can be afforded, we can reduce the average off estimation and Avg5WOE to 2.9% and 40%, respectively.

In our view, power consumption, rather than area, is the main problem in future processor's design. The ATD is accessed only when a task misses in the data or instruction L1 cache and moreover, only one entry is active at a time, thus its power consumption is low. We evaluate the area and power per access for the tag and data arrays of the three L2 cache configurations used in this Chapter as shown in Table 4.3.

Table 4.3: The power and area requirements of an ATD array in three L2 cache configurations. The power is measured in nanojoule, and the area is measured in square millimetre

	2MB		4MB		8MB	
	Power	**Area**	**Power**	**Area**	**Power**	**Area**
L2 Data Array	0.122	3.921	0.176	7.692	0.266	17.632
L2 Tag Array	0.005	0.118	0.006	0.233	0.007	0.464
Total L2 Array	0.127	4.04	0.181	7.926	0.272	18.095
ATD	3.98%	2.93%	3.15%	2.94%	2.56%	2.56%

To that end we use CACTI 6.5 [39] and assume a 32nm technology and sequential access to tags and data for power efficiency as it is common practice in L2 caches. We observe that the power and area requirements of the ATD array are always less than 4% in all configurations.

4.4.3 Memory Bandwidth Sensitivity

In our processor setups, the memory bandwidth is not identified as a main source of interaction between tasks. In order to show this point, we measure the memory bandwidth requirements of the evaluated workloads in all processor setups as shown in Figure 4.7.

We observe that 90% of the workloads require less than 8 GB/s bandwidth and that all of them require less than 12 GB/s. This is in line with latest DDR3 dual-channel memory that supports more than 15GB/s. In our processor setup and with the set of benchmarks we use the memory bandwidth is not an issue. In other setups with less cache or less memory bandwidth, the issue can be a problem. We leave this as part of our future work.

4.4.4 ITCA and Cache Partitioning Algorithms

Cache Partitioning Algorithms dynamically partition a shared cache among running tasks. Cache partitioning algorithms significantly improve metrics such as throughput [25, 49, 63], fairness [28, 38] and Quality of Service [25, 38].

An CPU accounting mechanism is also required in the presence of a cache partitioning algorithm, since tasks suffer slowdowns in their progress because they can only make use of a portion of the L2 cache. The cache partition changes dynamically, so the progress of the task (and hence the CPU time to account to it) also changes. Our

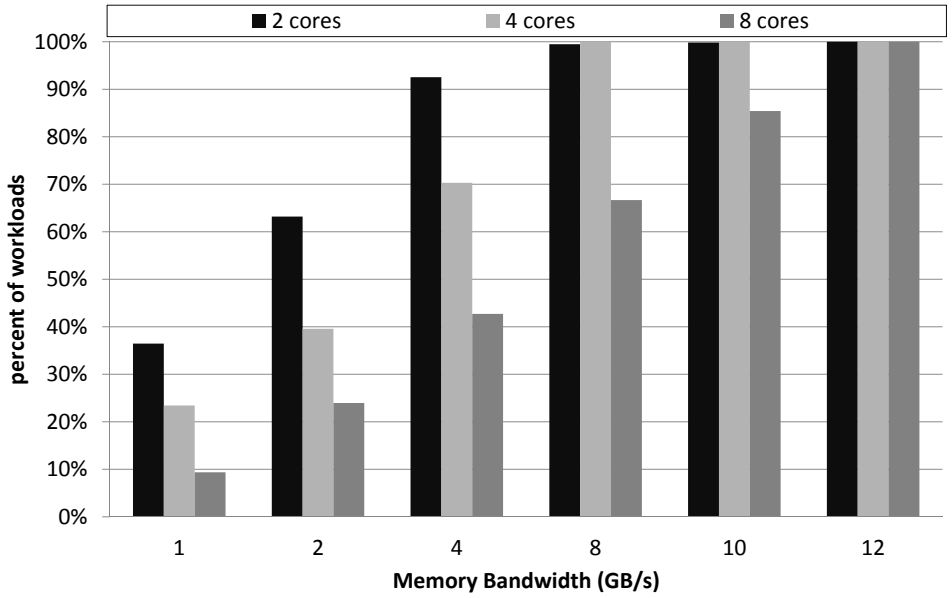


Figure 4.7: Memory bandwidth requirements for three CMP configurations

ITCA proposal can be applied to systems with cache partitioning algorithm without changes. The only conceptual difference is that the tasks do not suffer intertask conflicts as each task has a separate partition of the cache. However, we consider that a task is not progressing due to the cache partitioning algorithm when it suffers a miss in the L2 cache and a hit in its ATD. The ATD is already present in designs with cache partitioning algorithms and our CPU accounting mechanism can make use of it. In such a design, the only hardware cost of ITCA is the logic shown in Figure 4.4 (b).

In the previous sections, ITCA was evaluated on a CMP with a shared L2 cache with Least Recently Used (LRU) as replacement policy. The LRU scheme tends to give more space to the tasks that access more frequently to the cache hierarchy. Next, we evaluate the accuracy of ITCA when using a dynamically partitioned cache. Dynamic cache partitioning algorithms change the partition to adapt to the varying demands of competing tasks. We have chosen MinMisses algorithm [49], which attempts to minimize the total number of misses among all tasks sharing the cache and increase system performance.

For this study we compare the off estimations of the CA and ITCA on 2-, 4- and 8-core configurations. We use the same workload groups explained in beginning of Section 4.4.

Figure 4.8 shows the off estimation of the CA and ITCA with LRU and MinMisses

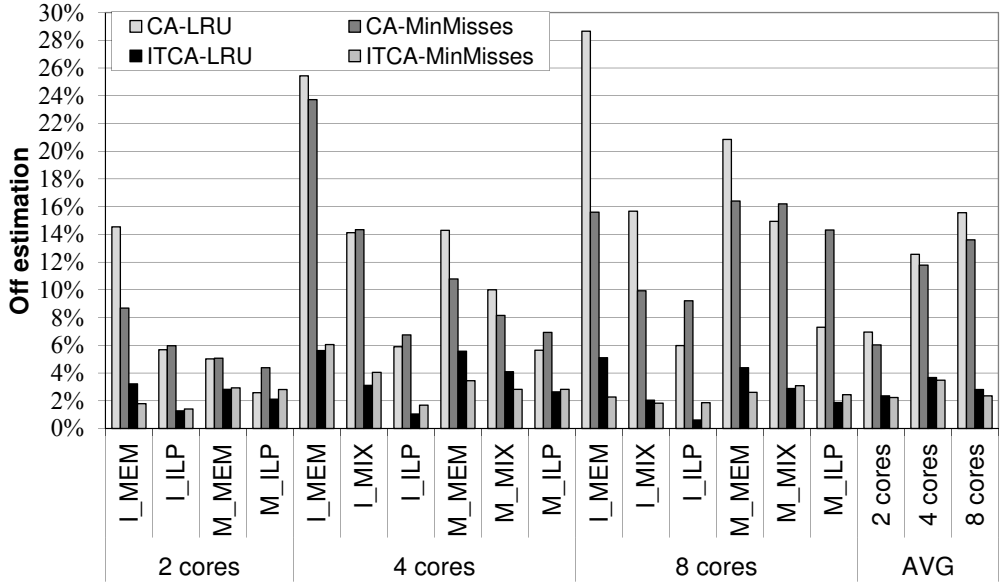


Figure 4.8: Off estimation of the CA and ITCA with LRU and dynamic cache partitioning algorithms, MinMisses

replacement policies. We observe that with LRU, the results are the same as the ones obtained in Section 4.4.1. The CA with LRU (denoted CA-LRU) obtains the worst results when the PTk task has high ILP and any of the STks is memory bound. ITCA combined with LRU (denoted ITCA-LRU) performs better than CA-LRU, reducing the off estimation to 2.4% (2 cores), 3.7% (4 cores) and 2.8% (8 cores) on average, as we observed in Section 4.4.1.

The CA presents a high variability in the off estimation when used in conjunction with MinMisses. MinMisses reduces the number of L2 misses and, consequently, the interaction between tasks. MinMisses assumes that all misses are equally important and tends to give more space to the tasks with higher L2 cache necessities, while harming the less demanding tasks. In some workloads, MinMisses cannot satisfy the cache necessities of the PTk, causing the PTk to suffer a lot of intertask misses, and increasing the off estimation. As a consequence, in some groups, the off estimation is high, reaching 24% and 16% off estimation in the group I_MEM in 4 and 8 cores.

ITCA-MinMisses provides much more stable results than CA-MinMisses. ITCA-MinMisses reduces the average off estimation of the CA-MinMisses from 6.0% (2 cores), 12% (4 cores) and 14% (8 cores) to 2.2% (2 cores), 3.5% (4 cores) and 2.4% (8 cores). In comparison with CA-MinMisses, ITCA-MinMisses consistently reduces the off estimation to less than 6.0% in all groups and all configurations.

To sum up, the combination of ITCA with dynamic cache partitioning algorithms significantly reduces the off estimation of the CA. Furthermore, ITCA leverages the ATDs already present in the MinMisses scheme with nearly no extra hardware addition motivating the use of both schemes simultaneously.

4.5 Improved ITCA Accounting

ITCA abstracts processor's architecture and takes into account only a few HRSIs as shown in Figure 4.4 (b). In particular, ITCA considers five different HRSIs: *RobEmpty*, *ITinstruction*, *RenameStalled*, *InterTopRob* and *AllInter* that work as follows:

1) *RobEmpty* indicates whether the ROB is empty and *ITinstruction* indicates whether a task has an intertask L2 instruction cache miss. If both are active (gate 1 in Figure 4.4 (b)), we stop accounting because there is an intertask L2 instruction miss and the machine is not utilized due to that.

2) *RenameStalled* detects if the register renaming stage is stalled and the signal *InterTopRob* indicates if there is an intertask L2 data cache miss at the top of ROB. If both are active, we know that the core is stalled due to an intertask L2 data cache miss, so we stop accounting.

3) *AllInter* determines if all the active entries in the MSHR contain an intertask L2 data miss, that is whether there are only intertask L2 data cache misses in flight (which corresponds to state (s2)), in which case ITCA stops accounting.

The way these signals are combined in order to determine whether or not to account to a task has been deduced from processor inspection: our basic premise was not to account cycles to a task when it suffers an intertask L2 miss that cannot be overlapped with any intratask L2 miss.

While ITCA mechanism is simple and intuitive, there might be hidden optimizations that may improve CPU accounting. In order to check whether our intuition is correct, we explore the complete design space with brute force. We analyze the accuracy results of all ITCA variants in which we *combine* the same HRSI in different ways. The best accounting scheme, denoted *Improved ITCA* (I^2TCA), will be the accounting scheme with the best accuracy among all possible combinations.

We start by identifying the percentage of time each HRSI determines the accounting decision. Our results show that the percentage of time that the ROB is empty due to an intertask L2 instruction miss is very low (0.05% on average and always less than 0.5%). Therefore, those cycles do not significantly affect the final accuracy of ITCA and the corresponding HRSIs (*RobEmpty* and *ITinstruction*) are not considered in

Table 4.4: Defined states of a task and accounting decision

HRSI states				Accounting decision	
No.	RenameStalled	InterTopRob	AllInter	ITCA	I^2TCA
7	1	1	1	0	0
6	1	1	0	0	0
5	1	0	1	0	1
4	1	0	0	1	1
3	0	1	1	0	1
2	0	1	0	1	1
1	0	0	1	0	1
0	0	0	0	1	1

our brute force approach. Hence, we have only three HRSIs affecting the accuracy of ITCA: *RenameStalled*, *InterTopRob* and *AllInter*.

Each row in Table 4.4 represents an *HRSI state*. A HRSI state is composed of three HRSIs: [*RenameStalled*, *InterTopRob*, *AllInter*]. Each HRSI can be 0 or 1, indicating whether this signal is active or not. For example, the state [$RenameStalled$, $InterTopRob$, $\overline{AllInter}$] corresponds to the HRSI state number 6, in which the signals *RenameStalled* and *InterTopRob* are active, while *AllInter* is not. In our exploration, we collect the amount of cycles a task spends in each of the eight HRSI states. An CPU accounting mechanism can either account that cycle to a task or not. This leads to a total of $2^8 = 256$ possible accounting schemes. For example, ITCA only accounts cycles: in HRSI state 0 [$\overline{RenameStalled}$, $\overline{InterTopRob}$, $\overline{AllInter}$], HRSI state 2 [$\overline{RenameStalled}$, $InterTopRob$, $\overline{AllInter}$] and HRSI state 4 [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$].

In order to evaluate the accuracy of all possible accounting schemes, we explore several processor setups: we vary the size of the L2 cache (2MB, 4MB and 8MB), keeping a constant associativity of 16 ways. We also study various CMP architectures with 2, 4 and 8 cores, which means that nine different processor configurations have been analysed.

We measure the sensitivity to each particular HRSI state decision. In three states, there is a significant difference in off estimation depending on the accounting decision: in HRSI state 4 [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$], and HRSI state 0 [$\overline{RenameStalled}$, $\overline{InterTopRob}$, $\overline{AllInter}$] we have to account always since the task is progressing. In contrast in HRSI state 7 [$RenameStalled$, $InterTopRob$, $AllInter$] we have to stop accounting. For the remaining five HRSI states (1, 2, 3, 5 and 6 in Table 4.4), the accounting decision is not so clear. Figure 4.9 shows the average off estimation for the remaining $2^5 = 32$ combinations in the 4-core configuration

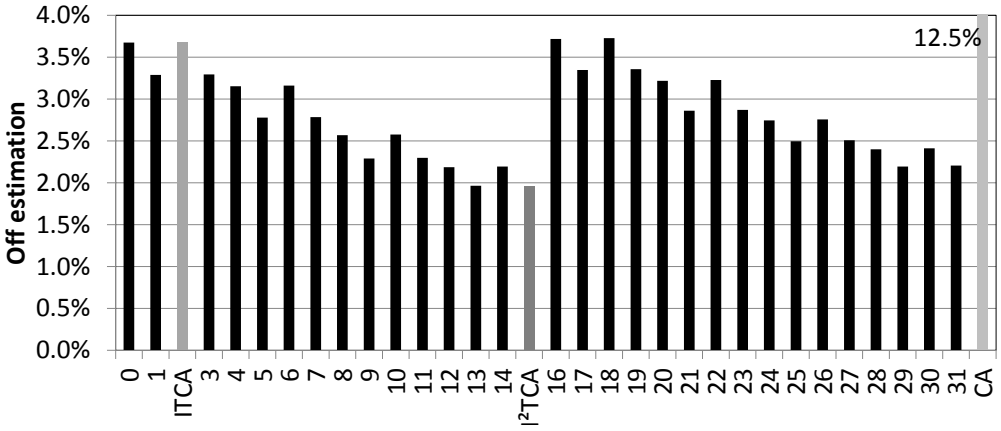


Figure 4.9: Average off estimation of all the combinations for 4 cores and a 16-way 4MB L2 cache

together with the CA. The same conclusions are derived for 2- and 8-core configurations. Accounting schemes are labelled according to the accounting decision taken in each of the five HRSI states. Thus, bar 0 shows the off estimation when in the remaining five HRSI states (1, 2, 3, 5 and 6) we stop counting. Bar 1 shows the off estimation when we count only in state 1, bar 2 when we count in state 2, bar 3 when we count in states 1 and 2, and so on and so forth. The CA has an average off estimation of 12.5%, while the 32 accounting schemes are under 4.0% off estimation. ITCA corresponds to the third accounting scheme in Figure 4.9. We observe that our original ITCA is quite close to the best observed combination, denoted *Improved ITCA* (I^{2}TCA). The average off estimation for ITCA is 3.7%, while for I^{2}TCA it is 1.96%.

The HRSI states in which ITCA and I^{2}TCA account cycles are shown in Table 4.4. We observe that both mechanisms account the cycles: in HRSI states 0 [$\overline{RenameStalled}$, HRSI state 4 $\overline{InterTopRob}$, $\overline{AllInter}$], [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$], and HRSI state 2 [$\overline{RenameStalled}$, $InterTopRob$, $\overline{AllInter}$]. In all these states, there are always intratask L2 misses overlapping with intertask L2 misses in the MSHR because the signal *AllInter* is not active (this was the base of our original intuition for ITCA). Hence, a task is accounted for those cycles. Moreover, ITCA and I^{2}TCA do not count the cycles in the HRSI state 7 [$RenameStalled$, $InterTopRob$, $AllInter$] and HRSI state 6 [$RenameStalled$, $InterTopRob$, $\overline{AllInter}$] because those states are not possible when a task runs in isolation. In other configurations, we conclude the same behaviour, and for this reason, we do not present all the results of the design space exploration.

The main difference between ITCA and I^{2}TCA is the accounting decision for state

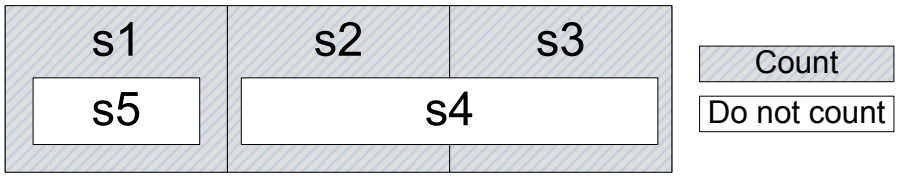


Figure 4.10: I^{2}TCA accounting decision for all possible states

(s2) explained in Section 4.3, in which there are only intertask L2 data misses in flight and the register renaming stage is not stopped. I^{2}TCA accounts cycles in this state, as shown in Figure 4.10, while ITCA does not, as shown in Figure 4.3. Though the task is not progressing at full speed, actual work is being done (register renaming is not stalled yet). Consequently, these cycles should be accounted to the task.

Regarding the logic to control the HRSI signals for I^{2}TCA, we observe that the signal *AllInter* is not needed to make a decision in the accounting and, hence, this signal can be removed from the logic. As a result, the I^{2}TCA logic is implemented with gates 1, 2 and 4 from Figure 4.4 (b) as shown in Figure 4.11.

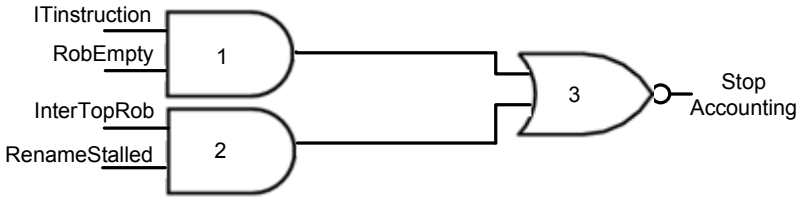


Figure 4.11: Logic to stop accounting required for I^{2}TCA

Next, we compare the average off estimation of the CA, ITCA, and I^{2}TCA in nine different processor configurations, as shown in Figure 4.12. We observe that the accuracy of the CA, ITCA and I^{2}TCA improve when we increase the size of the L2 cache with the same number of cores, since the number of intertask L2 misses diminishes. For instance, the off estimation of the CA in 4 cores with a 2MB L2 cache is 21%, but it is 6.2% with an 8MB L2 cache. The CA has higher off estimation than ITCA, and ITCA has worse accuracy than I^{2}TCA in all configurations. For example, in the configuration with 8 cores and a 2MB L2 cache, I^{2}TCA reduces the off estimation down to 4.5%, while the CA and ITCA present a 31% and 8.6% off estimation, respectively. Also, we observe that the off estimation of the CA increases when we vary from 2 to 8 cores with the same cache size per core. This is due to the fact that a task suffers more intertask L2 misses when the number of tasks running

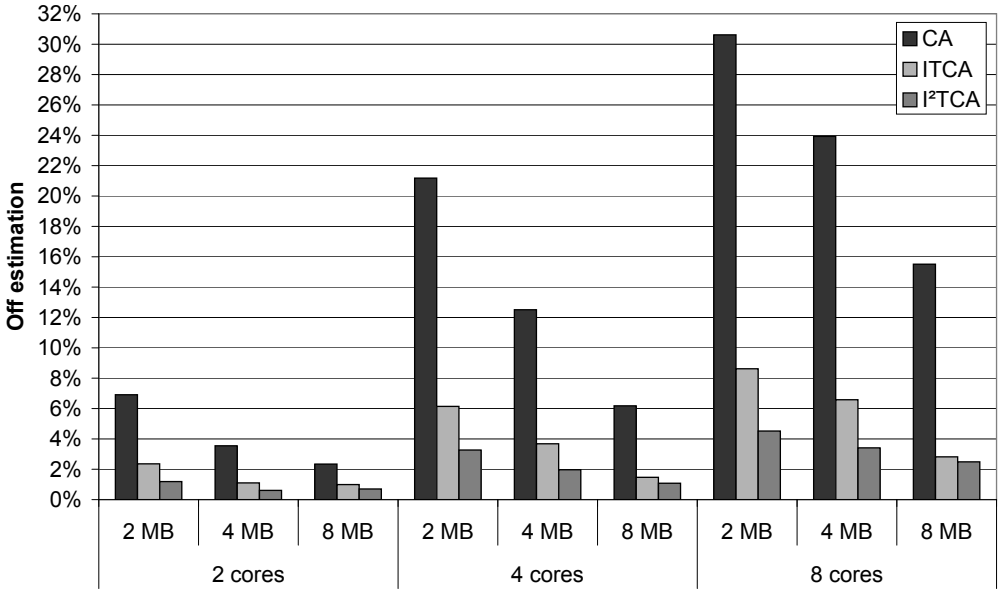


Figure 4.12: Average off estimation for 2, 4, and 8 cores and a 16-way 2MB, 4MB, and 8MB L2 cache

simultaneously increases. This behaviour is similar in I^{2}TCA, but the off estimation is always under 1.2% (2 cores), 3.3% (4 cores), and 4.5% (8 cores) on average.

I^{2}TCA reduces the average off estimation of ITCA, mainly in the five worst workloads (not shown in Figure 4.12): from 32% to 13% in the 2-core configuration, from 35% to 17% in the 4-core configuration, and from 20% to 14% in the 8-core configuration.

The design space exploration performed in this section confirms that ITCA provides reasonably accurate results. It also shows that with small changes we can implement an improved version of ITCA, denoted I^{2}TCA, which requires slightly less hardware support than the original ITCA, while clearly improving accuracy on a wide variety of experimental setups.

4.6 Even-Share Accounting

So far, we have assumed the full-share accounting in order to obtain the CPU accounting of tasks. In other words, the CPU time accounted to a task should be equal to the execution time of the task when it runs in isolation in the system with access to all the resources, and the task should be always accounted the same regardless of the workload in which it runs.

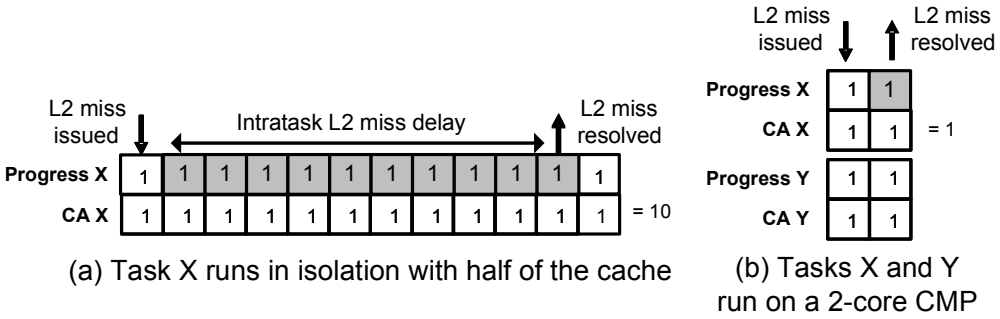


Figure 4.13: Synthetic example for explaining under-estimation with the CA and I^{2}TCA

In this section, we explore a different accounting approach, *even-share accounting*, which also accomplishes with the *principle of accounting*. We consider that a task should be accounted the CPU capacity it would require to run with its even part of processor resources, in other words, $1/Ntk$ of the shared processor resources, where Ntk is the number of tasks supported by processor. In the case of a CMP with a shared cache, a task should access to a $1/Ntk$ of the cache space. For example, assuming a two-core processor with 2 MB L2 cache, a task should only get access to 1 MB from L2 cache. This baseline accounting can be used together with recent cache management techniques developed for private last level caches [48]. We also show that few changes to I^{2}TCA are enough to adapt it to the even-share accounting without losing accuracy.

With full-share accounting, ITCA corrects the case in which the CA approach leads to over-estimation. In fact, this is always the case as the execution time of a task can only increase when it runs in CMP due to intertask conflicts.

With even-share reference accounting, the CA not only suffers from over-estimation, but it can also suffer from *under-estimation*. For example, assuming a given load misses in the L2 cache when a task X runs with $1/Ntk$ of the processor resources, the task might not have missed in the cache if it had shared the entire cache with another task in CMP mode. Consequently, task X might run faster in CMP than in the even-share processor setup, leading to under-estimation with the CA. To illustrate this situation in Figure 4.13, we use parameters of the synthetic example explained in Section 4.2.1, where let us 2-core in-order CMP processor and an L2 miss penalty of 10 cycles is assumed.

We refer to an *intertask L2 hit* to an access to the L2 cache that is a miss in the even-share configuration (with $1/Ntk$ of the L2 cache), and becomes a hit in the L2 cache when the task runs in CMP mode. This happens when the task runs in CMP

mode and it makes use of more than $1/Ntk$ of the L2 cache. In this case, the task can progress faster than when it runs in the even-share case, since the instructions after the intertask L2 hits are executed sooner in CMP mode. In other words, if an intertask L2 hit happens, a task would be able to execute more instructions than with $1/Ntk$ of the shared resource. In order to take into account this scenario, I^{2}TCA is adapted

The new decision of accounting 0, 1 or 2 cycles to a task in a given cycle depends upon whether or not the task makes the same progress it would do when running in its even-share processor setup. When a task goes slower than its execution with an even-share of the processor resources, the task is accounted 0 cycles; when the task is doing the same progress as its execution with an even-share of the processor resources, we account 1 cycle as usual; finally, when the task goes faster than its execution with an even-share of the processor resources, we account 2 cycles. This latter case happens when a task suffers an intertask L2 hit.

4.6.1 Hardware requirements

We add two new HRSIs to I^{2}TCA logic to cover the new cases in this scenario. The first HRSI, denoted *InterHit*, detects if there is an intertask L2 hit. We split the ATD among all cores in a CMP, assigning $1/Ntk$ to each core (remind that in the full-share accounting scheme, we assign an entire ATD to each core, while now we require only one ATD for all the N cores). An access that hits in the L2 cache and misses in the ATD is identified as an intertask L2 hit. Once an intertask L2 hit is detected, *InterHit* HRSI becomes active for the average latency of the misses in the L2 cache. This latency corresponds to the average time the miss would block the processor in single-threaded mode and is estimated in runtime. The second HRSI, denoted *IntraTopRob*, indicates if the oldest instruction in the ROB is an intratask L2 data cache miss. This signal is not useful with previous schemes, since in that state in the full-share accounting, the task progresses as in isolation (using all the processor resources).

Combining these signals with the ones defined for the original I^{2}TCA, we have the HRSI states shown in Table 4.5 in bold. In these new states in CMP, I^{2}TCA can account 0 cycles to a task if it progresses less than in isolation, 1 cycle if it progresses as in isolation, and 2 cycles if it progresses more than in isolation (during the average latency to memory of an L2 miss). Consequently, the accounting decision depends on the state of a task and its progress done.

The I^{2}TCA maintains the same accounting decision in the states explained in Section 4.5. The signal *IntraTopRob* cannot be active when signals *AllInter* or *InterTo-*

Table 4.5: States of a task and accounting decision

RenameStalled	InterTopRob	AllInter	**InterHit**	**IntraTopRob**	**I^{2}TCA**
1	1	1	0	0	0
			1	**0**	**1**
1	1	0	0	0	0
			1	**0**	**1**
1	0	1	0	0	1
			1	**0**	**2**
1	0	0	0	0	1
			0	**1**	**1**
			1	**0**	**2**
			1	**1**	**1**
0	1	1	0	0	1
			1	**0**	**2**
0	1	0	0	0	1
			1	**0**	**2**
0	0	1	0	0	1
			1	**0**	**2**
0	0	0	0	0	1
			0	**1**	**1**
			1	**0**	**2**
			1	**1**	**2**

pRob are active. In fact, *AllInter* indicates that there are no intratask L2 data misses in the pipeline while *InterTopRob* shows that there is an intertask L2 data miss at the top of the ROB. Therefore, *IntraTopRob* can only be active in the states [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$] and [$\overline{RenameStalled}$, $\overline{InterTopRob}$, $\overline{AllInter}$].

When the signal *InterHit* is active, a task is running faster than when it receives 1/N-th of the cache, and hence, I^{2}TCA accounts two cycles in all states except [$RenameStalled$, $InterTopRob$, $AllInter$], [$RenameStalled$, $InterTopRob$, $\overline{AllInter}$] and [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$]. In the former two states, a task is progressing slower than when receiving its even-share of the cache as *RenameStalled* and *InterTopRob* are active but, at the same time, it is going faster as *InterHit* is also active. Thus, we assume that both effects compensate each other and that the task progresses as fast as in the even-share case. Consequently, we account one cycle to the task. In the state [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$], the signal *IntraTopRob* is important to decide the accounting. In this state, if *InterHit* is active and *IntraTopRob* is not active, a task can execute instructions faster than in the even-share case and, consequently, I^{2}TCA accounts two cycles to it. In other situations in the same state [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$], the progress of a task in CMP mode is equal as in the even-share case and I^{2}TCA accounts one cycle to the task.

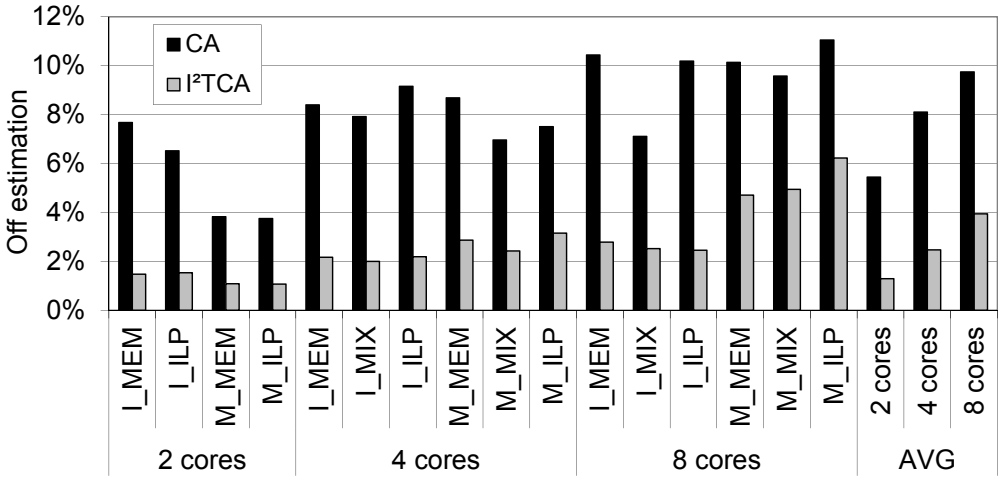


Figure 4.14: Off estimation of the CA and I^{2}TCA with the even-share accounting for 2, 4, and 8 cores and 16-way 2MB, 4MB, and 8MB L2 caches

4.6.2 Accuracy Results

Figure 4.14 shows the off estimation of I^{2}TCA and the CA for our three processor setups. We show the average results of each group as we described in beginning of Section 4.4. The bars labelled AVG represent the average of each CMP configuration for all the groups. We can see that the off estimation of the CA increases with the number of cores sharing the L2 cache, from 5.4% (2 cores) to 9.7% (8 cores). We also observe that the accuracy of I^{2}TCA is better than the accuracy of the CA in all groups. This is due to the fact that a task suffers both intertask L2 misses and intertask L2 hits during its execution. I^{2}TCA takes into account this situation and hence, on average I^{2}TCA reduces the off estimation down to 1.3% (2 cores), 2.5% (4 cores) and 3.9% (8 cores).

4.7 Other Considerations

4.7.1 Performance Counters

Intuitively, one could think that with the performance counters that are present in current processors we can accurately approximate the CPU capacity of each task in a MT processor. However, the main disadvantage of performance counters is that they do not measure the effect that one task can have on the other running tasks. For example, current performance counters report the number of L2 cache misses for a

given task when it runs together with other tasks. However, there is no information about the value of the same counters if the task had run in isolation, making it hard to derive the relative progress the progress did.

In our view, with the performance counters in current processors, we cannot provide an accurate estimation of the CPU capacity. In order to support this claim, we try to provide an accurate measure of the CPU capacity based on the events we can measure in our infrastructure. After studying several approaches, we use an approach that accounts each task X the time the workload executes in the MT processors, TR_{MT}, weighted by the percentage of instructions this task X executes, I_X, with respect to all the instructions executed in the workload. We denote this approach *Performance Counter Instruction-Based* (*PCIB*) accounting. For example, let's assume that we execute a workload composed of two tasks X and Y in a multi-core processor that executes I_X and I_Y instructions, respectively. In this model, we assume a linear relation between the number of executed instructions and the percentage of resources received. So, if with 100% of resources the processor executes $I_X + I_Y$ instructions, the processor uses $I_X/(I_X + I_Y)$ percentage of resources to execute I_X instructions.

Scaled Classical Approach (sCA): Another intuitive solution consists in accounting to each task X, $1/N$ of the time it is running in an MT processor with N is number of tasks supported by a processor. That is, $TA_X^{MT} = (1/N) \cdot TR_X^{MT}$. In this approach, we assume that each task receives an even part of the processor resources and that it makes $1/N$ of the progress it would do if run in isolation.

4.7.1.1 Results

Our results show that PCIB and sCA report an average off estimation of 48% and 47%, respectively for the 2-core configuration. For the same configuration, the Avg5WOE is 97% and 50%, respectively. For the 4-core configuration, both PCIB and sCA approaches report an average off estimation of 72%. The Avg5WOE is 98% and 75% respectively. For the 8-core configuration, both PCIB and sCA approaches report an average off estimation of 86%. The Avg5WOE is 99% and 87% respectively.

The sCA approach results in higher off estimations than the CA because CPU bound tasks can make a significant progress in CMPs (much more than $1/N$) as they only share the L2 cache with other tasks.

Regarding the PCIB approach, let us assume that we run a memory bound task as PTk and an ILP task as STk. In this situation, the STk executes many more instructions than the PTk, so the PTk is accounted a very small fraction of the time it is

running. However, in reality the PTk is making almost the same progress as in isolation since it is not suffering intertask cache misses. This introduces a significant error in the CPU accounting of the PCIB scheme.

4.7.2 Other Proposals Providing Fairness

Several hardware approaches deal with the problem of providing fairness in CMP processors. Although, fairness is a desirable characteristic of a system, it cannot be used to provide an accurate CPU accounting. There are two main flavours of fairness.

Several proposals approach fairness in MT processors by providing the *same amount of resources* to each running task. However, ensuring a fixed amount of resources to a task does not translate into a direct CPU capacity that can be accounted to that task [9, 25, 41, 50]. This is mainly due to the fact that the relation between the amount of resources assigned to a task and its performance can be different for each task. Hence, although all N tasks running in a MT processor receive 1/N of the resources, their relative progress is different, and so it should be their accounted CPU capacity.

Another set of proposals consider that an architecture is fair when all tasks running on that architecture make *the same progress*. For example, let's assume a 2-core CMP with running tasks X and Y. The system is said to be fair if the progress made by X and Y is the same in a given period of time: $P_X = P_Y$. However, the fact that $P_X = P_Y$ does not provide a quantitative value of the progress. Thus, the OS cannot account CPU capacity to each task according to their progress. In other words, having $P_X = P_Y$ does not provide any information about CPU accounting since P_X can be any value lower than 1. Hence, even if an architecture is known to be fair, it is not enough properly to account CPU capacity to each task.

In other approaches [12, 40], the goal is to provide fairness in the shared memory system through the control of the number of memory requests from each different task. These proposals achieve the same interference for each task in the memory hierarchy. It is not enough properly to account CPU capacity to each task.

For instance, in Figure 4.15 shows the progress of the PTk and the fairness of four different pairs of tasks measured as $fairness = 1 - \frac{(|P_X - P_{AVG}| + |P_Y - P_{AVG}|)}{2}$, where P_{AVG} is the average progress made by tasks X and Y. The fairness reaches its maximum value, 1, when the progress of both tasks is the same: $P_X = P_Y$. We observe that, for the two workloads on the left (*art*+twolf and *vpr*+vpr), the PTk (task in italics) does the same progress while the fairness is different in the two work-

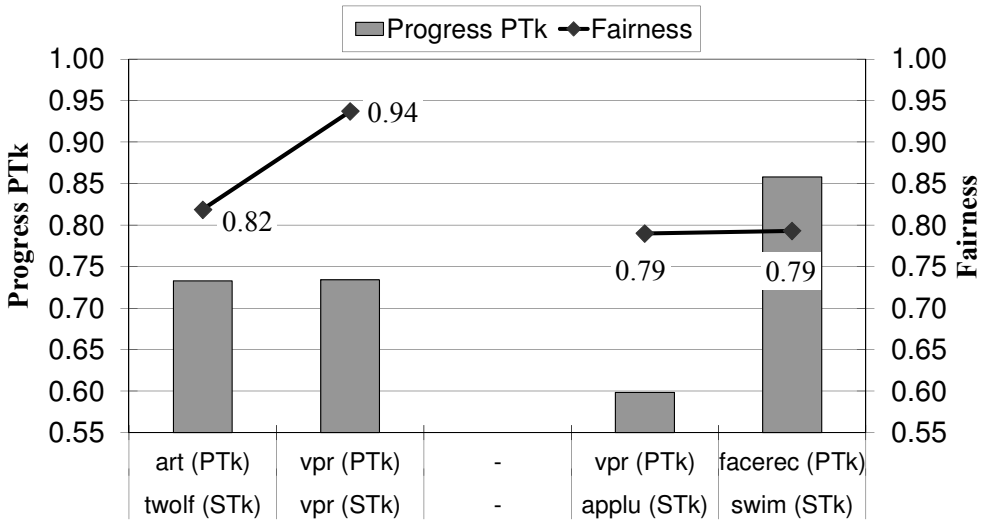


Figure 4.15: Fairness and progress of the PTk for four different workloads

loads. For the two workloads on the right (*vpr*+applu and *facerec*+swim), both workloads present the same fairness while the progress done by the PTk is different.

In short, multi-core systems that provide fairness do not necessarily provide accurate CPU accounting. Therefore, these systems require an accurate CPU accounting mechanism as well.

4.7.3 Parallel Tasks

In this Chapter, we have evaluated ITCA with multiprogrammed workloads (workloads composed of single-threaded tasks). ITCA also works, with minimal changes, for multi-threaded workloads. The interaction between threads in a parallel task can be *positive* when, for example, one thread prefetches data for another thread. This behaviour is intrinsic to the task, and hence it also occurs when running in isolation. When one multi-threaded task runs with other tasks it may suffer *negative* interaction, i.e. it may suffer intertask misses. ITCA already accounts for this situation, the only difference is that we have to track which threads belong to the same task, and do not consider a miss as an intertask miss when one thread evicts data from another thread of the same task. Only when two threads from different tasks evict each other's data, we report an intertask miss and stop the accounting if necessary. Consequently, a task identifier field has to be added to entries in the MSHR to detect intertask misses. This identifier has to be assigned to tasks by the OS, and provides to the architecture when scheduling is decided. Common OSs, such as Linux, already use an identifier for

the task (PID) and for the threads within the same task (TID). No other changes are required in the ITCA implementation.

With ITCA, the CMP processor reports a different accounting for each thread of a multi-threaded task through the Accounting Registers (AR) of each core. The OS will then combine the values of each AR into a single task figure, similarly to what currently happens for the TimeStamp Register for Intel architecture. The exact combination strategy is out of the scope of this study.

Notice that ITCA does not have to be aware of the synchronization among threads of a multi-threaded task. For example, if a thread is spinning on a lock and even if the multi-threaded task is not progressing during that time, the thread is using the processor, so it is accounted processor time.

4.8 Summary

CMP architectures introduce complexities when measuring CPU capacity utilized by tasks because the progress done by a task varies depending upon the activity of the other tasks running at the same time. The current accounting mechanism, the classical approach (CA), introduces inaccuracies when applied in CMP processors. In this Chapter, we have presented hardware support for a new CPU accounting mechanism called *InterTask Conflict-Aware* (ITCA) accounting with improved accuracy in CMPs. In 2-, 4- and 8-core CMP architectures, ITCA reduces the off estimation down to 2.4% (2 cores), 3.7% (4 cores) and 2.8% (8 cores) while the CA presents a 7.0%, 13% and 16% off estimation, respectively.

The combination of ITCA with dynamic cache partitioning algorithms significantly reduces the off estimation with respect to the CA. Furthermore, ITCA leverages the ATDs already present in many cache partitioning algorithms (such as the Min-Misses scheme [49]) with nearly no extra hardware addition, motivating the use of ITCA and cache partitioning algorithms simultaneously.

We have further improved the accuracy of the ITCA accounting mechanism without increasing its implementation complexity. We have seen that the accounting should be stopped when the register renaming is stalled due to an intertask miss that is at the top of the ROB. This Improved ITCA (I^2TCA) accounting nearly halves the off estimation of ITCA in all configurations. In an 8-core configuration, the off estimation is reduced from 6.0% to 3.5%. Finally, we showed the effects of a change in the accounting approach (even-share accounting) on the accuracy of different CPU accounting mechanisms.

Chapter 5

CPU Accounting for CMP+SMT Processors

In this Chapter, we analyse the accuracy of current CPU accounting mechanisms for SMT processors, and we show that they are not adequate for CMP+SMT processors. Consequently, we propose a novel CPU accounting hardware mechanism for CMP+SMT processors, denoted *Micro-Isolation Based Time Accounting* (MIBTA) which (1) increases the accuracy of accounted CPU accounting; (2) provides much more stable results over a wide range of processor setups; and (3) does not require tracking all hardware shared resources, significantly reducing its implementation cost. In particular, previous proposals lead to inaccuracies between 21% and 79% when measuring CPU capacity in an 8-core 2-way SMT processor, while our proposal reduces this inaccuracy to less than 5.0%.

5.1 Introduction

In processor microarchitecture designs, the general tendency [24] is the integration several TLP paradigms in a single chip to offer different benefits in exploiting TLP. As a result, these processor microarchitectures yield increasingly throughput with reduced cost and improved energy efficiency. For example, the Intel core i7 [51] and IBM POWER7 [56], are CMP+SMT processors which each core implements SMT.

An important aspect of CMP+SMT processors is improved hardware resource utilization. On a CMP+SMT processor, concurrently executing tasks can share costly hardware resources that would otherwise be underutilized such as on-core and on-chip resources. Higher resource utilization improves aggregate performance and enables lower-cost design. However, it also it also introduces complexities in the accounting

of CPU capacity of running tasks.

In CMP+SMT processors, the relation between the percentage of resources assigned to a task and the CPU accounting of the task is nonlinear. So as to overcome this situation, hardware support has been proposed to improve the way in which CPU capacity is measured in CMP [33, 34] as explained in Chapter 4 and SMT processors [13, 36] that will be analysed in Section 5.2. In both cases, the focus is on measuring the CPU capacity of a task taking account to interference of other tasks. However, these CPU accounting mechanisms are not accurate in CMP+SMT processors because they are not aware the interference either on-core or off-core together. For this reason, we present a new CPU accounting mechanism for CMP+SMT processor, called *Micro-Isolation Based Time Accounting* (MIBTA).

The rest of this Chapter is structured as follows. Section 5.2 analyses current CPU accounting mechanisms for SMT processors. Section 5.3 introduces our CPU accounting mechanism for CMP+SMT processors. Section 5.4 provides the experimental results, and Section 5.5 discusses other considerations regarding CPU accounting. Finally, Section 5.6 concludes this work.

5.2 Background

In the state of art, some CPU accounting mechanisms have been proposed for SMT processors. These mechanisms take into account hardware resource conflict in order to estimate dynamically the CPU accounting of each task. In addition, these mechanisms are implemented in our simulator and are evaluated for CMP+SMT processors in Section 5.4.6.

5.2.1 Processor Utilization of Resources Register

The Processor Utilization of Resources Register (PURR) [36, 18, 21, 22, 8] is a per-task CPU accounting mechanism designed by IBM and uses in the IBM POWER^{TM} processor family. The PURR mechanism is aware of the resource sharing and tries to approximate CPU accounting using the rate of cycles of decode given to each task. In other words, the PURR mechanism estimates the CPU capacity of a task based on the number of cycles the task can decode instructions. For instance, in a N-way SMT processor which can decode instructions from up to one task each cycle, the PURR accounts a given cycle to the task that decodes instructions that cycle. If no task decodes instructions on a given cycle, all tasks running on the same core are

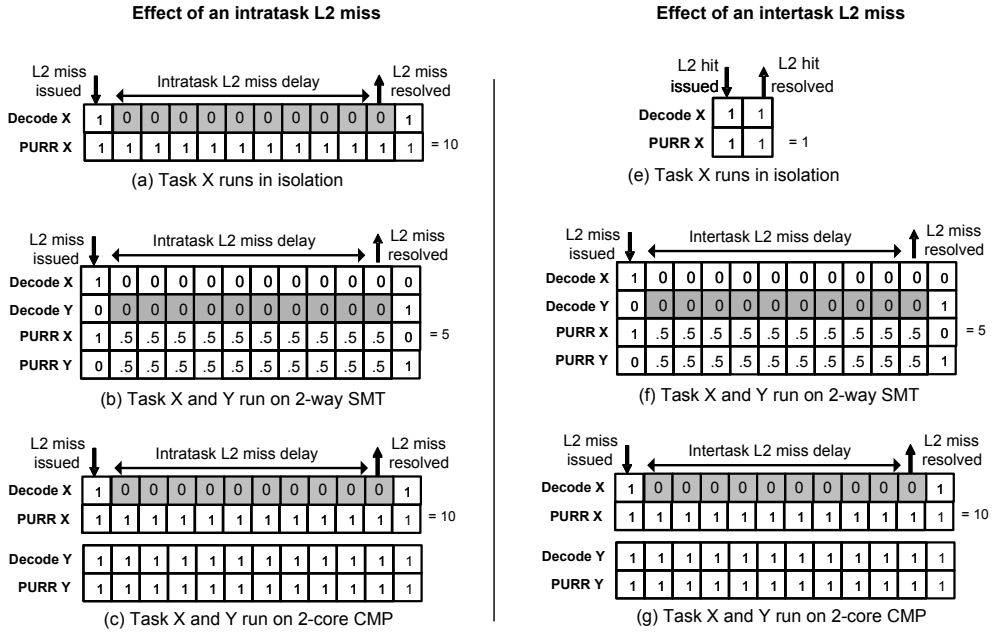


Figure 5.1: Synthetic example illustrates under-estimation and over-estimation with the PURR mechanism

accounted $1/N$ of the cycle, where N is the number of running tasks in the SMT.

In order to understand the PURR mechanism, we use parameters of the synthetic example explained in Section 4.2.1, but assuming each core supports 2-way SMT. The processor has different executing modes. In the example, we use the *SMT mode* to refer to the processor has an active core and other core is not utilized and two tasks executing onto the active core concurrently, whereas we use the *CMP mode* to refer to the processor has two active cores and a task executing in each core concurrently. In Figure 5.1, each square represents a processor cycle. The *Decode* row shows whether a task decodes instructions in each cycle or not. The values in the *PURR* row show the CPU accounting of each task.

Figure 5.1 (a) shows the situation when a given task, X, runs in isolation on the reference architecture and it experiences a (intratask) L2 cache miss. In this case, the processor is stalled just after the load miss is detected until the load resolves. Given that the task is alone running onto the processor, all decode cycles are accounted. The key point is that the task is accounted every cycle regardless of whether the processor is stalled or not. This accounting is correct as this L2 cache miss is intrinsic to the task X when running onto the architecture under consideration. In this situation, the task X is accounted ten cycles in order to execute this L2 cache miss.

Figure 5.1 (b) shows the same situation as in Figure 5.1 (a) but this time the task X runs with another task, Y, in the same core (SMT mode). The time the task X is accounted for processing an isolated intratask L2 miss is different in this case. While the L2 miss of the task X is blocking the processor, the task Y cannot decode instructions and, hence, the decode cycles are evenly accounted to both tasks[1]. Hence, the task X 'shares' its waiting cycles with the task Y, so the task X is charged only 5 cycles, which it is less what the task X should be accounted for a miss (ten cycles). Hence, when running in SMT mode, the cycles in which the processor is stalled are evenly charged to all tasks running onto the processor. When tasks that continuously stall the processor in isolation, their CPU accounting are under-estimation in SMT mode as their waiting cycles are charged to tasks running onto the processor. Normally, tasks with very low IPC are most of the time stalled for some resource conflict, hence, they are expected to have under-estimation when running in SMT mode. On the other hand, high-IPC tasks suffer from over-estimation as they are charged cycles that they would not have been charged, if they had run in isolation (the waiting cycles of the low-IPC tasks).

Figure 5.1 (c) shows the same situation that in Figure 5.1 (b) but this time the task Y runs in the second core (CMP mode). As the PURR mechanism was thought for SMT processors, we observe that the PURR and Decode behaviour of the task X is the same as in isolation, regardless whether or not the task Y is running on the second core. Hence, the under-estimation case does not arise in this processor configuration. In both cases, the task X is accounted ten cycles to process the L2 cache miss.

In Figures 5.1 (e)(f)(g), we show the situation where a task X suffers an intertask L2 miss due to the execution of another task Y in the processor. Figure 5.1 (e) shows the situation in which a task X runs in isolation and executes a memory access that hits in the L2 cache. In this situation, the memory access resolves in one cycle, so the task X is accounted only one cycle for processing the memory access.

Figure 5.1 (g) shows the situation in which a task Y executes in the second core (CMP mode). In this case, we assume that the task Y evicts the data of the task X from the L2 cache, causing the previous L2 hit of the task X to become an intertask L2 miss. This intertask miss causes the task X to stall its execution until it is resolved, ten cycles later, so in this case, the task X takes ten cycles to process the memory

[1] In an out-of-order processor before the processors stops, it experiences a transient state in which the hardware resources are progressively filled with instructions of task X [27] and both task X and task Y can continue running. However, the steady state in which the processor is stalled is still the longest, so at some point the tasks X and Y eventually stop.

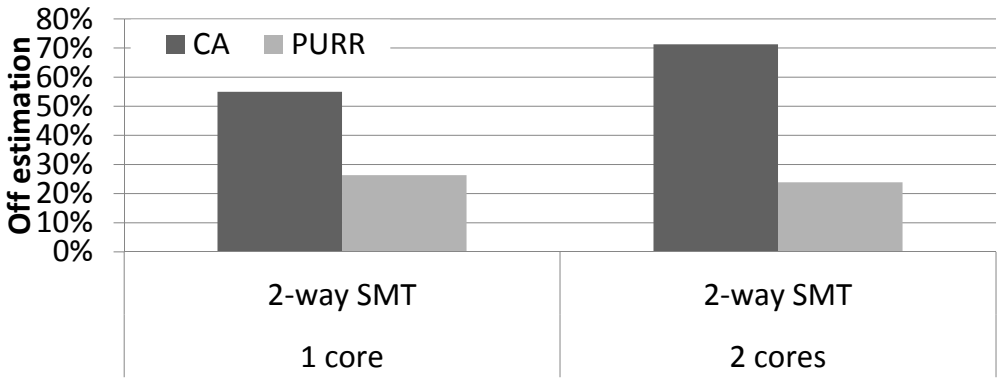


Figure 5.2: Measured CPU accounting accuracy for the CA and PURR mechanisms on a POWER5 processor. The lower the better

access.

Figure 5.1 (f) shows the case in which a task Y runs onto the same core with the task X (SMT mode). This case is similar to the case shown in Figure 5.1 (g), the only difference is that the time the task X is waiting the miss to be solved is shared between both tasks. Hence, with the PURR mechanism, the over-estimation of the task X is less in an SMT processor than in the case of a CMP processor because the cycles in which the processor is stalled are evenly charged to all tasks running onto the processor.

To illustrate the accuracy of the CA and PURR mechanisms, we use run several 2- and 4-task workloads from SPEC CPU 200 benchmarks on a POWER5 processor. We measure the CPU accounting of each workload and we the average in Figure 5.2. We observe that PURR significantly improves the accuracy provided by the CA, reducing its inaccuracy from 55% to 26% in the single core configuration. In the 2-core configuration, the CA shows even higher inaccuracies (reaching 71%) while PURR results are more stable (close to 24%). Even if the results of PURR are more accurate than the CA, there is still room for improvement to narrow down CPU accounting inaccuracies.

In conclusion, PURR mechanism represents a better solution in SMT processor than CA because it takes account the on-core intertask conflicts in its estimation. However, the PURR mechanism has the same results as CA mechanism in CMP processor due to off-core intertask conflicts. As a consequence, the PURR and CA mechanisms account over cycles per task in CMP processor as shown in Figure 4.2.

PURR mechanism has been adapted for the techniques pipeline throttling and Dynamic Voltage and Frequency Scaling (DVFS). This improved PURR mechanism

is denoted *scaled PURR* (SPURR) [17, 21, 22, 8]. SPURR provides a scaled count that compensates the impact of throttling and DVFS, and whose count is mathematically equivalent to

$$SPURR = PURR * (\frac{f_{effective}}{f_{nominal}}) * (1 - \frac{cycles_{throttled}}{cycles_{total}}) \quad (5.1)$$

where $f_{effective}$ is the effective frequency during the count-accumulation interval, $f_{nominal}$ is the nominal frequency of the processor that is adhered to in the absence of power management actions, $cycles_{throttled}$ is the number of processor cycles that the pipeline was halted during the count-accumulation interval, and $cycles_{total}$ is the total number of processor cycles during that interval.

5.2.2 Processor Utilization Recording Register

The Processor Utilization Recording Register [4] is a similar mechanism to PURR. The main difference with PURR is that the target SMT processor in [4] is able to decode instructions from up to two tasks per cycle, while the target of PURR is SMT processor can only decode instructions from one task per cycle.

This mechanism measures the portion of useful capacity allocated to each task, based on sampling dispatch information. In other words, the focus is put on the number of instructions a task dispatches. For example, if in a given cycle the first task dispatches 5 instructions and the second 2, the first task will be charged 5/7 and the second 2/7.

In short, the processor utilization recording register has the same behaviour as PURR and CA mechanisms in CMP processors.

5.2.3 Cycle Accounting Architecture

The Cycle Accounting Architecture (CAA) was proposed by Eyerman and Eeckhout [13], and is based on estimating the CPI stack [15] per task.

The CAA tracks fifteen different components of the CPI stack with dedicated hardware (front-end miss event table (FMT)), and also, estimates the increase in the number of per-task miss events due to sharing in SMT execution (cache and TLB misses, and branch mispredictions) as shown in Figure 5.3. This solution provides detailed information of the execution of each task at the cost of more complex structures (to track all possible events), logic and dedicated floating-point ALUs. Furthermore, the authors report 7.2% and 11.7% average prediction errors for 2- and 4-task

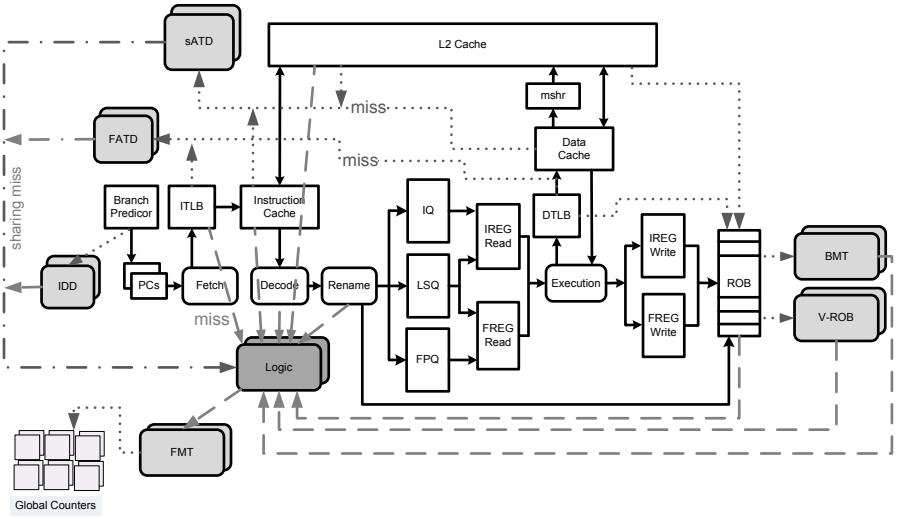


Figure 5.3: Logic and hardware required by CAA

workloads, respectively.

The CAA classifies the execution cycles of each task into three possible groups:

- *base* cycle: the task dispatches instructions in the right path. The task is making progress.
- *miss event* cycle: it consists of cache misses cycles, TLB misses cycles, branch misprediction cycles, and others event cycles. In these cycles, the task cannot dispatch instructions or dispatches in the wrong path, due to miss events (cache misses, TLB misses, and branch mispredictions). The task is not progressing.
- *waiting* cycle: the task cannot dispatch due to another task and hence, the task is waiting to dispatch new instructions because of SMT execution.

The CAA quantifies the reduction in per-task memory-level parallelism (MLP), estimating the MLP in isolation using a back-end miss event table (BMT) per task. Furthermore, the CAA calculates on correction factor per task in each interference of branch predictor, caches and TLBs with dedicated hardware. These hardware resources are a Sampled Auxiliary Tag Directory (sATD) [49] for set-associative caches, a Fully-Associative Tag Directory (FATD) for fully-associative caches and an ID Directory (IDD) for branch predictor as shown in Figure 5.3. In addition, in order to detect a long-latency data miss, the CAA estimates when ROB per task would be full in isolation using a Virtual ROB (V-ROB). In conclusion, the CAA estimates per-task

ST execution as the sum of the base cycles and miss event cycles with corresponding correction factors.

To calculate the different cycles, the CAA is based on the notion of a dispatch slot (in other words, a 4 wide dispatch processor has 4 dispatch slots per cycle). The CAA counts the number of dispatch slots per task in each cycle, classifying each slot in base, waiting and miss event. For instance, the base cycles are calculated dividing the number of base slots to wide dispatch processor.

Cycle Accounting Logic: When a task can dispatch an instruction in some slot, if the instruction is in the correct path, the base counter is increased. If the task dispatches an instruction from the wrong path, the slot has to be assigned to a branch misprediction counter. Since branch mispredictions are detected after some cycles, the authors propose to store the slot counters per branch and update the global counters when the branch is committed. In the case the branch was mispredicted, all the slots are assigned to the branch misprediction counter until instructions from the correct path are dispatched.

When a task cannot dispatch an instruction in some slot, this is due to two cases: (1) other tasks are using all dispatch slots, the waiting counter of the task is increased. (2) if the task suffers a miss, the corresponding miss counter is increased. The cycle accounting keeps track of several possible miss sources: instruction L1 cache, L2 or L3 instruction cache, instruction TLB miss, full ROB (due to L2 or L3 data misses, data TLB misses, long latency units, dependencies, etc.). When several back-end misses arise, the CAA gives priority to the miss associated to the first instruction in the ROB. When several front-end misses arise, a branch misprediction and an instruction L1, L2, L3 or TLB miss, the branch misprediction has more priority (only the branch misprediction counter is increased). If front-end and back-end misses occur in the same time, the front-end miss has more priority unless the ROB is full. In this situation, the back-end miss counter is increased.

5.3 Micro-Isolation Based Time Accounting

In this section, we introduce the *Micro-Isolation Based Time Accounting* (MIBTA) for CMP+SMT processors. To begin with, we focus on MIBTA for SMT processors where we study the on-core interference. Next, we analyse the off-core interference for CMP processors.

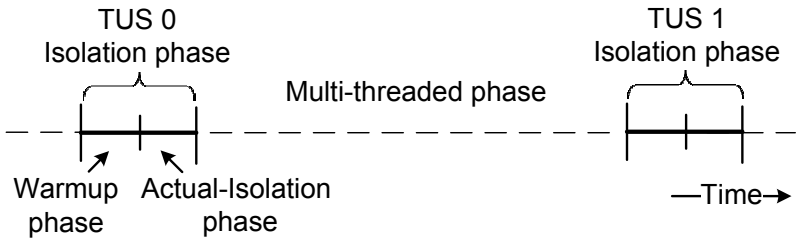


Figure 5.4: The isolation and multi-threaded phases in MIBTA mechanism

5.3.1 MIBTA for SMT Processors

Tracking each resource utilization in an SMT processor introduces high hardware cost, complicates its design and is architecture dependent. Moreover, accounting mechanisms do not directly contribute to improve system performance, which motivates the use of an accounting mechanism as simple as possible. In order to estimate the CPU capacity in SMT, we introduce *Micro-Isolation Based Time Accounting* (MIBTA) mechanism. Unlike previous proposals, MIBTA does not track task interaction in each shared resource in an SMT processor. MIBTA divides the execution of running tasks into two phases that are executed in alternate fashion, as shown in Figure 5.4.

- *Isolation (isol) phase*: During this phase, a task running on the core, denoted Task Under Study (TUS), is given access to all shared resources, and the other tasks are temporarily stalled. As a result, we obtain an estimate of the current full speed of that TUS during this phase which we call the *isolation IPC*. Note that the *isol* phase has to be kept as short as possible to reduce system performance degradation.

- *Multi-threaded (MT) phase*: During this phase, all tasks are allowed to run and their IPCs are also measured.

In subsequent *isol* phases, a task on a core become the TUS, and hence, its IPC is measured in isolation. In other words, the isolation phase of tasks is not consecutive and is active in different time in order to measure isolation IPC of tasks. For instance, assuming that two tasks X and Y are running onto a two-way SMT processor, both tasks start to execute in MT phase. In the first isolation phase, the task X becomes the TUS and its isolation IPC is measured. After the isolation phase finished, both tasks are running in MT phase again. In the second isolation phase, the TUS is the task Y,

and its isolation IPC is measured. Next, both tasks are running in MT phase again. In the third isolation phase, the task X becomes the TUS again, and this sequence is repeated during the execution of both tasks.

Even if the TUS is run in isolation, it may still suffer intertask conflicts in shared resources as in the precedent *MT* phase all tasks used those shared resources. In our simulated architecture, as described in Chapter 2, there are the following shared core resources[2]: fetch and issue slots, issue queue entries, physical registers, caches, TLBs, and the branch predictor. Private per-core first level instruction and data caches, TLBs, and the branch predictor can suffer destructive interference because an entry given to a task can be evicted by another task before it is accessed again. In order to get more insight into this interference, we have measured the average intertask conflicts the TUS suffers during the *isol* phase. We have observed that, as the *isol* phase progresses, the TUS evicts all data from other tasks. Consequently, the number of conflicts goes toward zero for the instruction cache, data cache, TLBs, and the Branch Target Buffer (BTB). We observed that 50,000 cycles after the beginning of the *isol* phase, most interference in these shared resources is removed. The branch predictor, Pattern History Table (PHT), takes much longer to clear: We have measured that it takes more than 5 million cycles before misses due to the interaction with other tasks (intertask misses) have disappeared. However, this interference is mostly neutral, giving a negligible loss in the branch predictor hit rate of less than 1%. Hence, we ignore the interference in the branch predictor.

To remove intertask conflicts in all core shared resources, we propose to split each *isol* sample into two subphases. During the first subphase, the *warmup phase*, that consists of 50,000 cycles, the TUS is given all resources, but its IPC is not measured. In the second subphase, the *actual-isolation phase*, the TUS keeps all resources, and its IPC is measured. The duration of this subphase is 50,000 cycles. In Section 5.4, we study the accuracy of MIBTA with different warmup and actual-isolation phase lengths.

During each actual-isolation phase, we count the number of instructions executed by the task X under study, $I_{isol,X}$. Dividing $I_{isol,X}$ by the number of cycles of the actual-isolation phase, we obtain a sample of the IPC in isolation of the TUS, $IPC_{isol,X} = \frac{I_{isol,X}}{cycles\_actual-isolation\_phase}$.

[2]Inter-core resource conflicts such as LLC or memory bandwidth contention are considered in the next section for CMP processors.

5.3.1.1 Hardware Implementation

The implementation of MIBTA requires reduced hardware support. In fact, many current processors already incorporate similar support that could be used to provide MIBTA's required functionality. MIBTA needs a countdown timer that is programmed to trigger at the end of each phase: warmup, actual-isolation and MT phases. The three registers that save these values, *wuReg*, *aiReg*, and *mtReg*, can be made visible and writable from the operating system.

MIBTA also requires that only one task runs during each *isol* phase. This can be done in a straightforward way by stopping the fetch of instructions of the other tasks in the core. Processors such as the IBM POWER7 already incorporate hardware thread priority mechanism [56]: When a hardware thread is assigned the lowest priority it is allowed to fetch instructions only once every dozen of cycles. MIBTA would simply require another priority level in which a task is not allowed to fetch further instructions until its priority is changed back to its previous value.

After MIBTA has stopped the fetch of instructions for a given task, task's in-flight instructions will eventually commit before the end of the warmup phase. The only information that the task would have in the core is its program counter and the architectural state registers. In processor architectures where the architectural registers are kept in a different register file than the physical registers, MIBTA can be implemented just with the small change in the fetch stage mentioned above. In processors in which the architectural and physical registers share the same register file (the most common case), it is necessary to deallocate from the register file the architectural registers of the non-running tasks in the isolation phase so that the TUS enjoys as many resources as when actually running in isolation. We do this, with small changes in the pipeline. In particular, we propose a mechanism that releases architectural registers of non-running tasks and locks them into the LLC cache. We denote this solution *Register File Release* (RFR) mechanism. A similar approach is implemented in the Intel Sandy Bridge processor [51], where the state of the machine in a given core can be flushed to the LLC cache to turn off the core and reduce system energy. This operation takes in the order of hundreds of cycles, and deploys the data path already implemented in the processor, not requiring extra wires for data. Overall, MIBTA works as follows:

- Just after the MT phase ends (and the warmup phase starts), we stop fetching instructions from all the tasks except the TUS. When there are no instructions from the other tasks in the ROB, we store the information from the architectural registers of the other tasks in the LLC cache, locking the corresponding cache

lines. The time at which the other tasks have no in-flight instructions can be determined by deploying performance counters present in many current architectures that are able to measure it. Afterwards, the architectural registers from other tasks are released, increasing the number of renaming registers available to the TUS.

- At the end of the warmup phase, the number of instructions executed is saved into the register *wuInstr*.
- When the actual-isolation phase ends, the instructions executed are saved into another register, *aiInstr*. The number of instructions executed during the actual-isolation phase is $I_{isol,X} = aiInstr - wuInstr$. All architectural registers are loaded from the LLC cache back into the register file, and the normal MT phase begins.

The information to store is 64 architectural registers per task (32 integer and 32 floating point registers). Every register is 64 bits, and hence we need to store 4096 bits per task (512B). In a 2- and 4-way SMT processor, the total storage required is 0.5KB and 1.5KB, respectively. Since in our processor configuration the LLC cache line size is 128B, we only need 4 cache lines per task. In all configurations, we have measured that less than 1,000 cycles are required to release all the architectural registers and lock them in the LLC cache. In our simulation infrastructure, we simulate this process in detail.

5.3.2 MIBTA for CMP processors

In addition to the conflicts on on-core resources, the main source of interference in a CMP+SMT processor is the shared off-core as such LLC cache. When measuring the intertask conflicts the TUS suffers once it enters in an *isol* phase, we observed that the interference in the LLC cache extends for several million cycles. These intertask misses give rise to a significant performance degradation (more than 30% for some benchmarks), leading to a bad estimation of the IPC of the task in isolation and, consequently, of its CPU accounting. The long duration of LLC conflicts makes the solution of extending the warmup phase infeasible, as it would introduce significant performance loss.

In order to overcome this problem, MIBTA has to detect every time the TUS suffers an intertask LLC miss and take into account this information in the final CPU accounting of the task. MIBTA uses the hardware provided by ITCA mechanism

explained in Chapter 4. MIBTA detects an intertask LLC miss by using an ATD and tracks the intertask LLC miss by utilizing the MSHR.

The ATD is a large structure, and consequently, reducing its overhead without decreasing the accuracy of the proposed CPU accounting mechanism is a crucial objective. A first possibility consists on eliminating the ATDs, relying on the warmup phase to bring to the LLC cache a significant part of the task's data. However, as mentioned earlier, several million cycles are required to eliminate all intertask LLC cache misses. Thus, these misses will be accounted as intratask misses during the actual-isolation phase and the accuracy of the CPU accounting mechanism will be affected.

A second option consists on considering a sampled version of the ATD, denoted sATD [33, 49], which only monitors some sets of the LLC cache and obtains the miss rate in isolation. Under this approach, we track intertask misses to the sampled sets. However, when the number of sampled sets is reduced, accuracy significantly decreases since we are not detecting intertask misses to non-monitored sets.

Based on the fact that sampled ATDs are very accurate predicting LLC miss rates [49], we propose to track the probability of having an intertask miss in the sampled sets, i.e. the ratio between intertask misses and total misses to the sATD of the task. During *MT* phase, the number of intertask and total misses per task are tracked and accumulated in two registers per task. When the *isol* phase begins, the ratio between these values is computed and stored as a 10 bit integer that is multiple of $\frac{1}{1024}$ (0 represents 0, 512 represents 0.5, 1023 represents 0.99, and so on and so forth). This threshold is always computed during warmup phase. During actual-isolation phase, the same intertask miss probability is assumed for accesses to non-monitored sets. When missing on a non-monitored set, a 10-bit random number is generated with a Linear Feedback Shift Register (LFSR) [6]. If this number is less than the previously obtained threshold, this LLC miss is predicted to be an intertask miss. Otherwise, LLC miss is an intratask miss. This Randomized version of the Sampled ATD, denoted `RSA`, predicts intertask misses to these non-sampled sets. In Section 5.4, we present a detailed study of the accuracy of all these possible implementations, concluding that `RSA` provides nearly the same accuracy as the entire ATD with the same hardware cost as a sampled ATD.

5.3.3 CPU Accounting in MIBTA

To determine whether the task is progressing in a given cycle of an actual-isolation phase, we make use of the decision provided by I^{2}TCA mechanism explained in Chapter 4. This mechanism is specifically developed for accounting in CMP architectures with shared caches. I^{2}TCA stops accounting to a task in two situations: (1) when the ROB is empty because of an intertask LLC cache instruction miss, and (2) when the instruction in the top of the ROB is an intertask LLC miss and the register renaming stage is stalled. In other situations without intertask LLC misses or when intertask misses overlap with intratask misses, the accounting is not stopped since the task is performing similar progress it would make in isolation.

The cycles accounted to each task, and the instructions it executes during the actual-isolation phase are accumulated into special purpose registers per task, denoted *Isolation phase Cycles Register* (*ICR*) and *Isolation phase Instructions Register* (*IIR*), respectively. We also accumulate the instructions and cycles tasks are running in the MT phase into the *MT phase Instruction Register* ($MTIR$) and the *MT phase cycle Register* ($MTCR$), respectively.

These registers are read-only like the *time stamp register* in Intel architectures, and can be communicated to the OS. On every context switch, the OS reads for each task X the ICR_X, IIR_X, $MTIR_X$ and $MTCR_X$ registers. With this information, the OS estimates the time to account to each task as: $TA_{X,I_X} = ICR_X + \frac{IPC_{MT,X}}{IPC_{isol,X}} \cdot MTCR_X$, where $I_X = IIR_X + MTIR_X$ is the total number of executed instructions, the IPC in isolation $IPC_{isol,X} = \frac{IIR_X}{ICR_X}$ and the IPC as part of the workload $IPC_{MT,X} = \frac{MTIR_X}{MTCR_X}$. At the context switch boundary, in fact on every clock tick, the OS also updates metrics of the system and carries out the scheduling tasks. Thus, the OS could potentially use the information provided by MIBTA to find better co-schedulers, similarly to [16]. When a task is swapped out, its associated ICR_X, IIR_X, $MTIR_X$, $MTCR_X$ can be updated in the *task struct* and are reset before the next task starts.

5.4 Evaluation Results

We perform several studies to evaluate the accuracy of MIBTA. First, we focus on a single core processor to determine the best design parameters for MIBTA accounting mechanism. Afterwards, we evaluate different implementations of our proposal that minimize the storage overhead of MIBTA without decreasing its accuracy. Finally,

Table 5.1: Simulation Configuration

Core configuration				
	2-way SMT		4-way SMT	
Number of core	1,2,4,8		1,2,4	
Issue Queue entries	48 int, 48 fp, 48 ld/st		64 int, 64 fp, 64 ld/st	
Physical Registers	164 int, 164 fp		256 int, 256 fp	
ROB size	256		352	
Execution Units	4 int, 2 fp, 2 ld/st			
Fetch Policy	ICOUNT 1.8			
Branch predictor	2K entries, gshare			
Branch Target Buffer	256 entries and 4 ways			
Clock Frequency	2.0GHz			
Cache/Memory Configuration				
Core/s	1	2	4	8
LLC (shared)	2MB	4MB	8MB	16MB
	16 ways, 8 banks, 128 Bytes			
Instruction (per core)	64 KB, 4 ways, 1 bank, 128 Bytes			
Data (per core)	64 KB, 8 ways, 1 bank, 128 Bytes			
ITLB (per core)	128 entries, 8 KB page			
DTLB (per core)	256 entries, 8 KB page			
Latencies	LLC (15), Memory (300)			

we evaluate MIBTA in a CMP+SMT configuration with two, four and eight cores, and compare its results with previously proposed CPU accounting mechanisms.

To evaluate the accuracy of MIBTA, we make use of several processor setups shown in Table 5.1: four different core counts (one, two, four, and eight), and 2- and 4-way SMT cores, for a total of seven different configurations[3]. We feed our simulator with traces collected from the whole SPEC CPU 2006 benchmark suites (see Chapter 2 for more details). Running all N-task combinations is infeasible as the number of combinations is too high. Hence, we randomly choose a workload for each benchmark in the SPEC CPU 2006 suite, generating a total of 26 workloads for each evaluated configuration.

5.4.1 Sensitivity Analysis for Single-Core Architectures

We determine the design parameters that provide the best trade-off in terms of accuracy and throughput in a single core processor. When moving to a CMP+SMT scenario, similar results will be obtained, as we show in Section 5.4.4.

Figure 5.5 shows the off estimation and throughput degradation results for a 2- and 4-way SMT processor under different sampling intervals, ranging from 1.3 to 20.8 million cycles. The sampling interval is the number of cycles between two *isol* phases

[3]We do not simulate the 8-core 4-way SMT configuration due to simulation time constraints

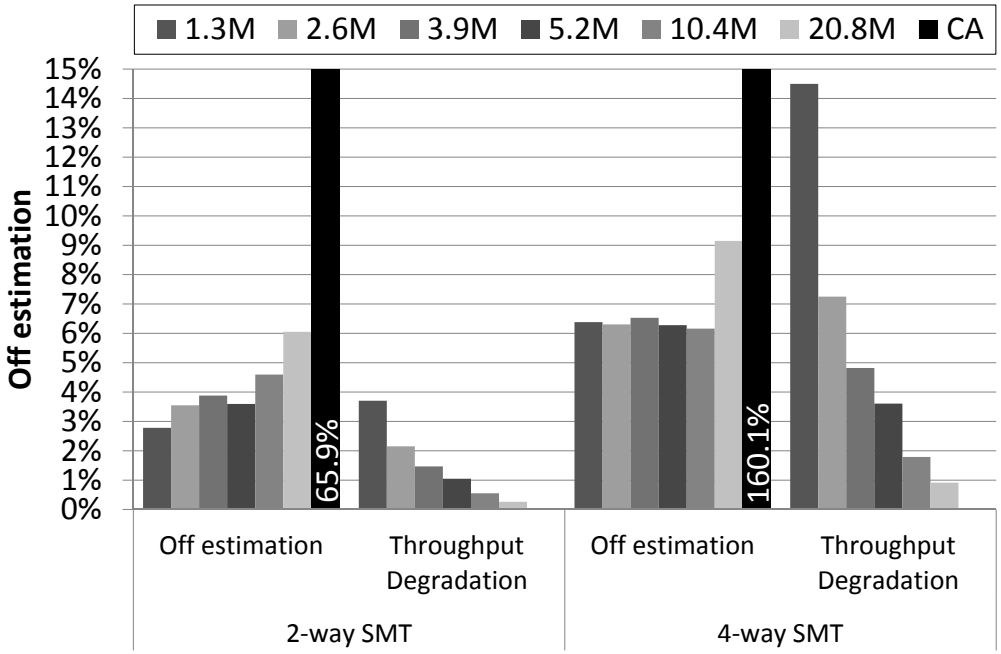


Figure 5.5: MIBTA off estimation and throughput degradation on an SMT processor under different sampling intervals

of a given task. In both configurations, the off estimation increases with the sampling interval, since MIBTA cannot capture some phase changes of the task. In contrast, the throughput degradation decreases with the sampling interval, since there are less *isol* phases that degrade total throughput. With all sampling intervals, the off estimation of MIBTA clearly improves over CA. With the shortest sampling interval, off estimation reaches just 2.8% and 6.4% in 2- and 4-way SMT processors, respectively.

Throughput degradation is significant for a sampling interval of 1.3 million cycles, but decreases with higher sampling intervals, at the cost of worse off estimations. When choosing a sampling interval of 5.2 and 10.4 million cycles for 2- and 4-way SMT, an interesting trade-off between off estimation and throughput degradation is obtained: 3.6% and 6.2% off estimation, and 1.0% and 1.8% throughput degradation, respectively. This corresponds to a period between any two *isol* phases of 2.6 million cycles. For the remaining experiments, we maintain this value, as it represents a good balance between accuracy and performance.

Figure 5.6 shows the average of MIBTA off estimation for length of the isolation phase from 60 to 100 thousand cycles with different lengths of warmup phase (WP) and actual-isolation phase (AIP). In general, the best results are obtained with

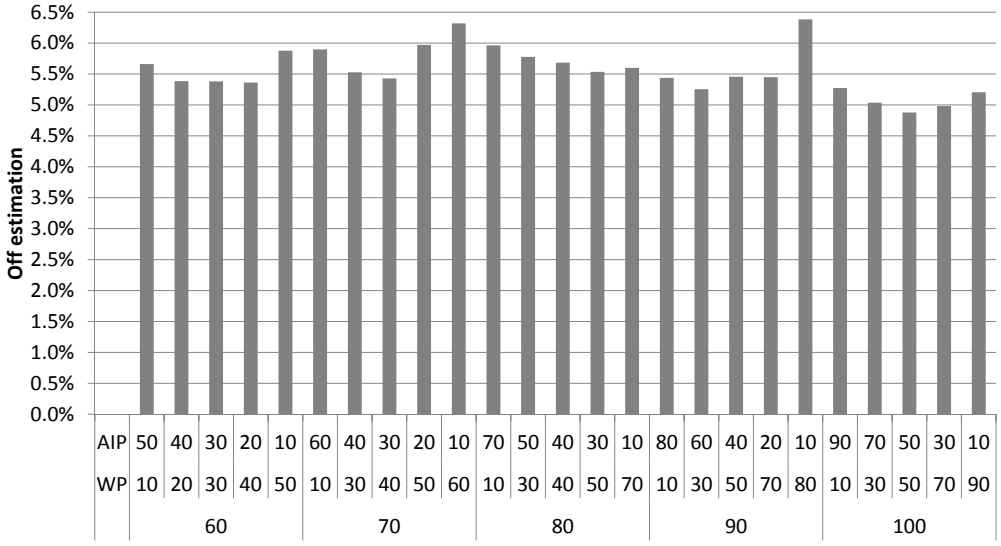


Figure 5.6: Average off estimation of MIBTA for isolation phase lengths from 60 to 100 thousand cycles with different lengths of warmup phase (WP) and of actual-isolation phase (AIP)

balanced warmup and actual-isolation phases. In contrast, the worst results are obtained with extreme values, when either the warmup or the actual-isolation phase is only 10 thousand cycles. On average for 2- and 4-way SMT processors, the optimal results are obtained with warmup and actual-isolation phases of length 50 thousand cycles with an average 4.9% off estimation. Several configurations are close to this optimal value, with the worst results (6.4% off estimation) obtained with warmup and actual-isolation phases of 80 and 10 thousand cycles, respectively. In the remaining experiments, we maintain 50 thousand cycles for both warmup and actual-isolation phases.

Next, we explore which resource conflicts lead to the off estimation obtained with MIBTA on the SMT processor. When comparing the execution of each task in isolation (the task runs alone in the system) and in the actual-isolation phase in our scenario, we detect that LLC conflicts are the key contributor to the off estimation of MIBTA. Under an SMT configuration with a perfect LLC cache (without any intertask LLC cache conflicts), the average off estimation is reduced to 4.0%. The remaining off estimation is explained by the sampling interval, since MIBTA can not capture all task phases.

Finally, we discover that the architectural state of all stopped tasks that is stored in the register file affects the accuracy of the estimated IPC for each TUS in the isolation

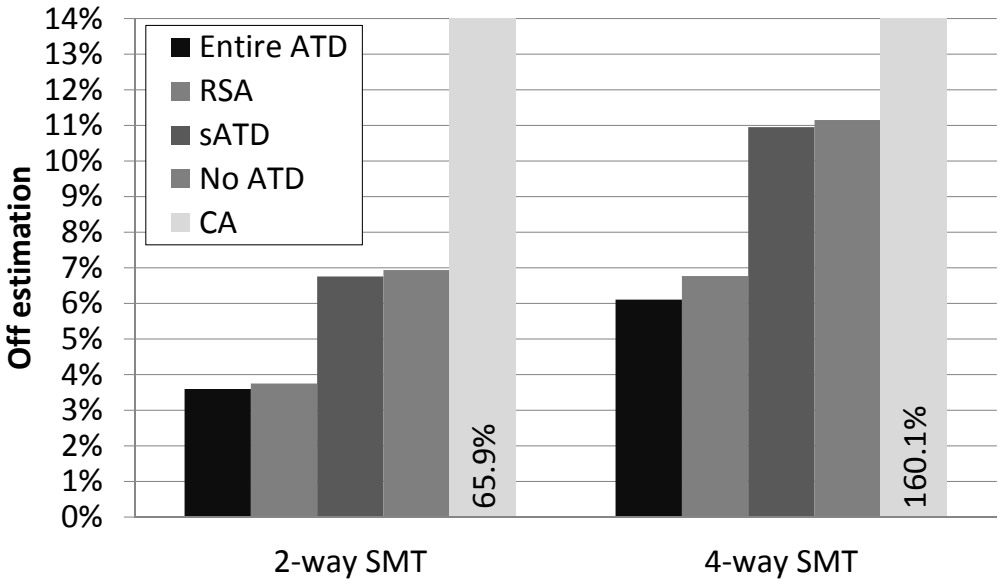


Figure 5.7: MIBTA off estimation in 2- and 4-way SMT processors using different storage overheads

phase. In the isolation phase, the task under study can not make use of the registers that store the architectural state of all stopped tasks. In the 4-way SMT configuration (where four tasks run simultaneously), this effect is high, and hence, this finding motivates the use of the register file release mechanism that we evaluate in Section 5.4.3.

In conclusion, MIBTA accurately works with a period of 2.6 million cycles and an isolation phase of 100 thousand cycles. In this configuration, MIBTA does not significantly degrade the system throughput.

5.4.2 MIBTA Storage Overhead

Next, we evaluate different implementations of MIBTA with different storage overheads devoted to tracking intertask LLC misses. Figure 5.7 shows the off estimation results of the four possible implementations in 2- and 4-way SMT processors. Without using any storage (`no ATD` configuration in Figure 5.7), MIBTA reduces off estimation from 113% to just 9.0% on average. The warmup phase effectively eliminates the majority of intertask conflicts and, even if some intertask LLC misses are not detected, the off estimation is heavily reduced.

When considering 32 sampled sets out of the 1024 sets of the LLC cache (`sATD` configuration), the storage overhead supposes just 960 bytes per task. However, very few intertask misses are detected and, consequently, off estimation is very close to the

configuration without ATDs. In contrast, when tracking the intertask ratio with the randomized sampled ATD (RSA configuration), the off estimation is reduced to 5.3% on average, very close to the 4.9% average off estimation with the entire ATD. Implementing RSA requires a sampled ATD with 32 sampled sets, two 64 bits registers, two shifter registers, an LFSR, and four 64 bits special purpose registers. In our current configuration, the total storage overhead per task is 1KB. At core level, one bit per entry in the ROB and the MSHR are required, as well as three 20-bit registers. This supposes an extra 0.04KB and 0.05KB in 2- and 4-way SMT cores, respectively.

Since characteristics of tasks dynamically change, intertask miss rate should reflect these changes. However, we also wish to maintain some history of the past *MT* phases. Thus, after the *isol* phase ends, we multiply all the values of the intratask and intertask misses times $\rho \in [0, 1]$ in all tracking mechanisms. During the MT phase, we keep accumulating these values for all tasks. Large values of ρ have larger reaction times to phase changes, while small values of ρ quickly adapt to phase changes but tend to forget the behaviour of the task. Small off estimation variations are obtained for different values of ρ ranging from 0 to 1 (less than 0.5% on average for the worst case), with the best results for $\rho = 0.5$. Furthermore, this value is very convenient as we can use a shifter to update the values. For all the experiments, we maintain this value.

5.4.3 Shared Register File

As mentioned previously, to further improve the accuracy of MIBTA in SMT processors, we need to take into account the contention in the shared register file. Each task has 32 architectural registers stored in each shared register file, which impacts the performance of the task under study in the actual-isolation phase, even if the other tasks are not running. Since there are 256 shared registers per register file in a 4-way SMT configuration, the task under study will get at most 160 registers ($256 - 3 \cdot 32$). As a result, it will suffer more contention in the register file than in isolation.

Figure 5.8 shows the results in a 2- and 4-way SMT processor setup when using the MIBTA mechanism with and without the register file release mechanism. In this experiment, MIBTA is combined with the RSA. The average error in these configurations is reduced to 3.2% and 5.2%, respectively. The off estimation reduction is significant in the 4-way SMT configuration, since the contention in the shared register file is much higher than in the 2-way SMT configuration (132 and 160 available register out of 164 and 256, respectively). In fact, the accuracy in the 4-way SMT con-

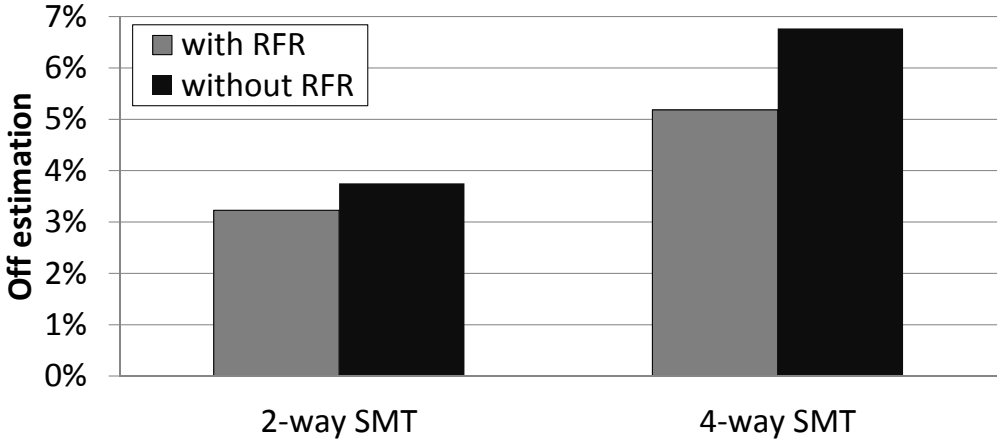


Figure 5.8: Accuracy with/without the register file release (RFR) mechanism

figuration is 23.5% better than without the RFR mechanism (5.2% instead of 6.8%). In all configurations, the extra throughput degradation due to this mechanism is insignificant (less than 0.05%).

5.4.4 MIBTA on CMP+SMT Architectures

Next, we move to a CMP+SMT scenario, with up to 8 cores sharing the LLC cache and memory hierarchy. In this case, during each isolation phase only one task is running on each core, removing on-core intertask conflicts after the warmup phase. However, there will be still some intertask conflicts when accessing the LLC and the memory hierarchy. Figure 5.9 shows the off estimation results for the seven different configurations. MIBTA has an off estimation under 5.0% in all 2-way SMT processors, while for 4-way SMT processors, the off estimation is always between 5.2% and 7.5%. MIBTA obtains better results than using the entire ATD due to the RFR mechanism and with a much lower hardware overhead. When using a sampled ATD, a solution with similar hardware overhead, the off estimation quickly raises to 18.5% and 15.6% in 4-core 4-way SMT and 8-core 2-way SMT configurations, respectively. In contrast, MIBTA has a more stable accuracy, suggesting that the LLC contention is correctly addressed by MIBTA.

Note that MIBTA takes into account bus, bank and memory controller conflicts indirectly when intertask misses are in-flight. However, bus, bank and memory controller conflicts are not considered by MIBTA when there are only intratask conflicts. In other words, the time in order to resolve an intertask miss may be increased due to the interference in bus, bank and memory controller. As MIBTA does not account

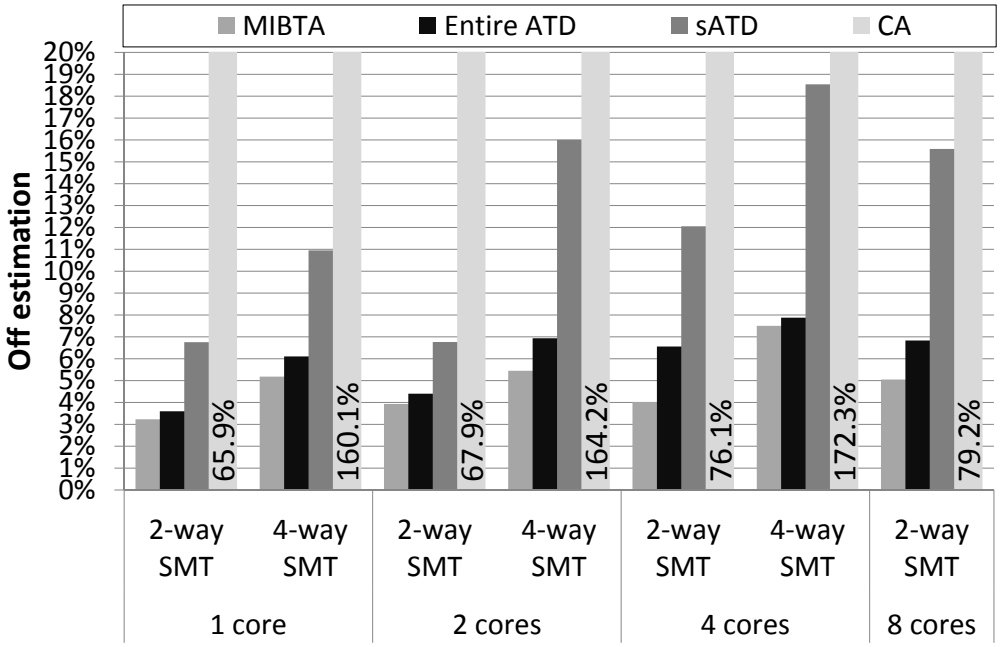


Figure 5.9: MIBTA off estimation for 7 different CMP+SMT configurations

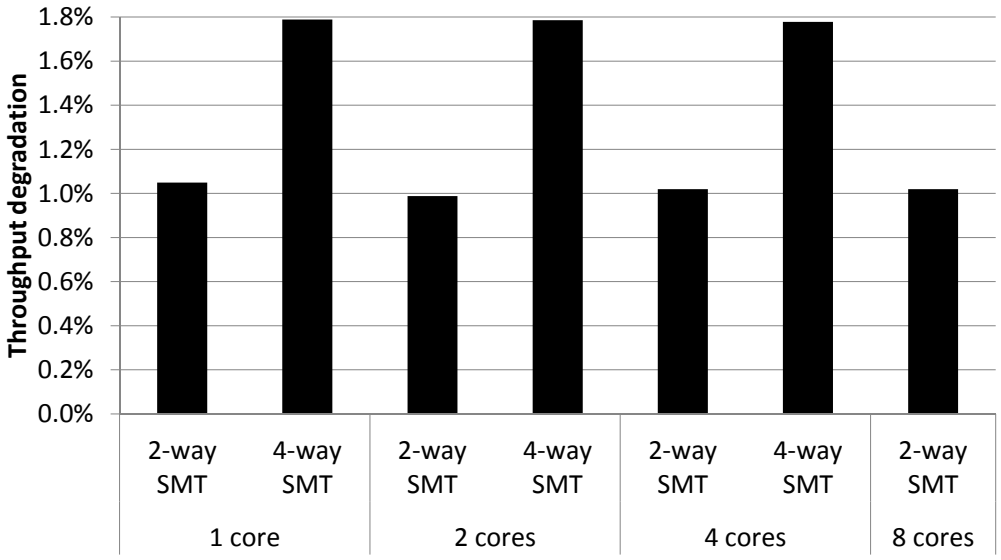


Figure 5.10: MIBTA system performance degradation for 7 different CMP+SMT configurations

the extra time due to intertask misses for CPU accounting of a task, the interference in these resources is considered in CPU accounting. In contrast, if the time to resolve an intratask miss is increased due to the interference in these resources, MIBTA does

not consider this situation, and hence, bus, bank and memory controller conflicts are not considered in this situation. Our results confirm that the effect of those conflicts is small on CPU accounting accuracy, making it not worthy to devote extra hardware to track them. We elaborate more on this point on Section 5.4.5.

As mentioned before, the throughput degradation of MIBTA basically depends upon the number of tasks that are stopped in each core during isolation phases. This behaviour is shown in Figure 5.10. We observe that the throughput degradation is 1.0% in 2-way processor and 1.8% in 4-way processor. The increase of off estimation from 1.0% to 1.8% is due to the number of stopped tasks during isolation phases. We also observe that the throughput degradation is nearly constant when we vary from 1 to 8 cores with the same number of SMT threads per core. In short, all values are below 1.8% and do not increase with the number of cores in the system.

5.4.5 Memory Bandwidth Sensitivity

As mentioned earlier in this section, the memory bandwidth has not been identified as a main source of interaction between tasks in our different processor setups. To illustrate this point, we measure the memory bandwidth requirements of the evaluated workloads in all processor configurations. Figure 5.11 shows the percentage of workloads that require a given memory bandwidth to reach their maximum performance. Note that in all our processor setups, we keep the same ratio of LLC cache per core: 2MB/core.

We observe that the memory bandwidth requirements increase with the number of tasks simultaneously running on the system. For example, workloads in a 1-core 2-way processor only require 2GB/s to reach maximum performance, whereas the required memory bandwidth is almost 12GB/s in a 4-core 4-way processor. On average, we have measured that 90% of the workloads have a bandwidth requirement of less than 10GB/s in our setup. More importantly, all of them require less than 16GB/s to reach their maximum performance. This is in line with latest DDR3 dual-channel memories that support more than 15GB/s.

To sum up, we conclude that the memory bandwidth is not a problem in our processor setups and with the set of benchmarks we have used. In other setups with less cache or less memory bandwidth, memory bandwidth can be an issue. Consequently, we leave dealing with memory bandwidth contention in MIBTA as future work.

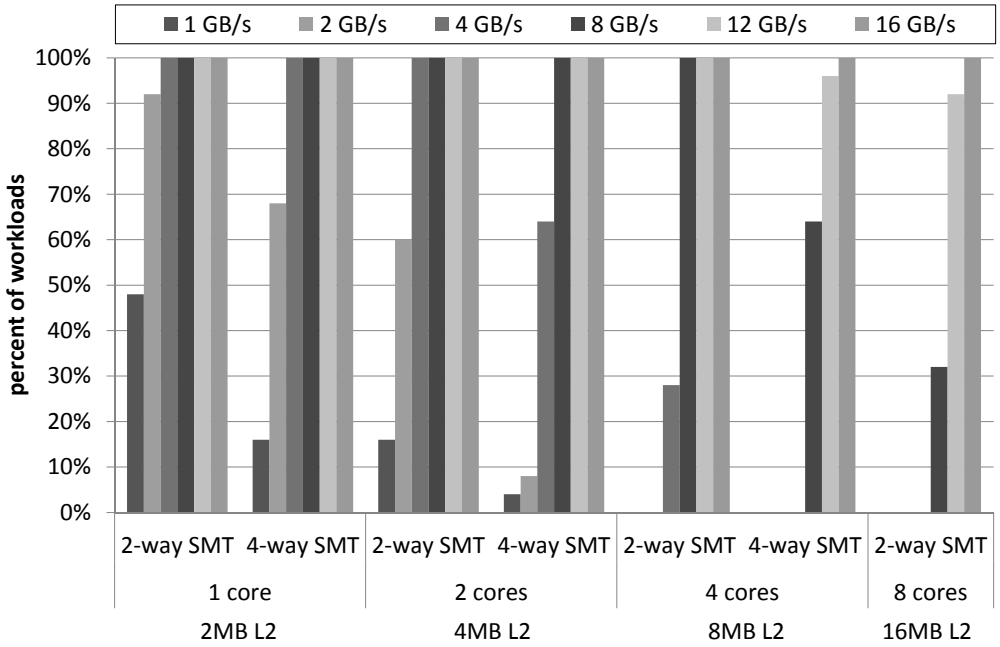


Figure 5.11: Memory bandwidth requirements for 7 different CMP+SMT configurations

5.4.6 Comparison with Other Accounting Mechanisms

Next, we compare the accuracy of MIBTA with previously proposed CPU accounting mechanisms. These mechanisms, described both in Chapter 4 and in Section 5.2, are also implemented in our simulator. Figure 5.12 evaluates the accuracy of the CA, ITCA, PURR, CAA and MIBTA across multiple processor configurations. The CA shows the worst results 112% average off estimation, since it is not aware of any intertask conflict. ITCA also has similar results since it was designed for CMP systems and does not take into account intertask core conflicts, with an average off estimation of 102%.

In the case of PURR, off estimation is always between 25% and 38%. PURR estimates CPU capacity based on the decode cycles of each task. When decode stage is stalled, the decode cycle is evenly split among tasks. This mechanism presents two sources of inaccuracy. First, when a task decodes, the other tasks are also progressing (this is one of the main motivations for SMT processors), but only the first task is accounted. And second, when a particular task stalls the processor due to a long latency miss, waiting cycles are accounted to all tasks.

CAA obtains more accurate results than PURR, specially for one core processors since it was originally designed for SMT processors. The off estimation ranges from

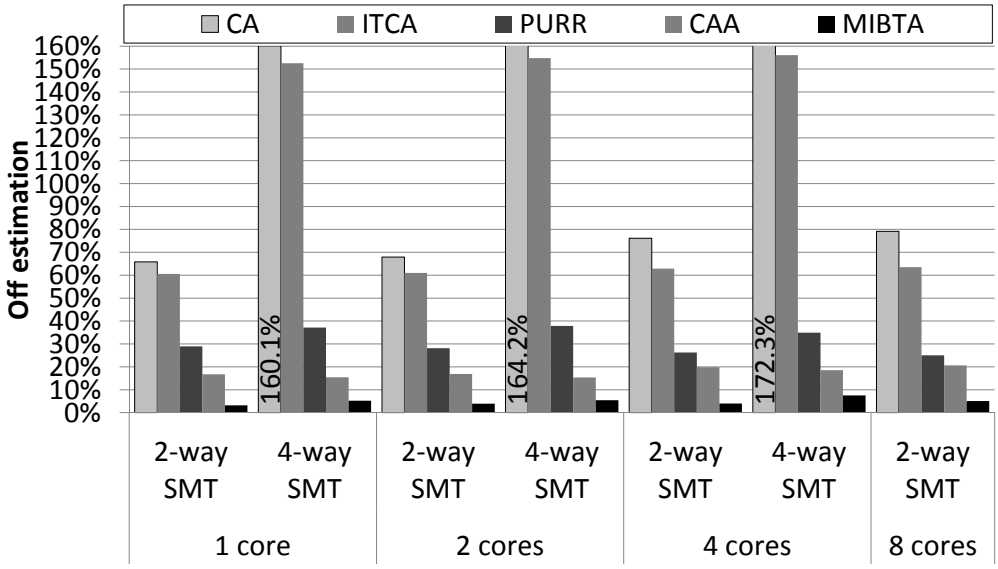


Figure 5.12: Off estimation with different accounting mechanisms across several processor configurations

16% in the single core configuration, to 21% in the 8-core configuration. This proposal assumes that the ROB is the main bottleneck for performance of an out-of-order architecture. For that reason, authors track the ROB occupancy in isolation and detect when the ROB would be full in that situation. However, authors do not consider the contention in other important resources such as the issue queues, the register file (authors assume that architecture registers are separate from rename registers), cache banks and memory bandwidth contention. When the number of cores increases, bank and memory bandwidth congestion become more significant and, as a result, the off estimation values get worse.

In contrast, MIBTA shows much more accurate and consistent results across all configurations, with average errors between 3.2% and 7.5%. The results shown in Figure 5.12 indicate that it is important to consider other shared resources such as the issue queues or the register files. Thus, a CPU accounting mechanism such as MIBTA leads to more accurate results independently of the processor configuration in which it is run.

Even if previous CPU accounting mechanisms do not suffer any performance degradation (or negligible), the experiments in Section 5.4.4 show that the performance degradation introduced by MIBTA is very low: less than 1.8% in all configurations. In terms of implementation cost, the CA and PURR require negligible hardware sup-

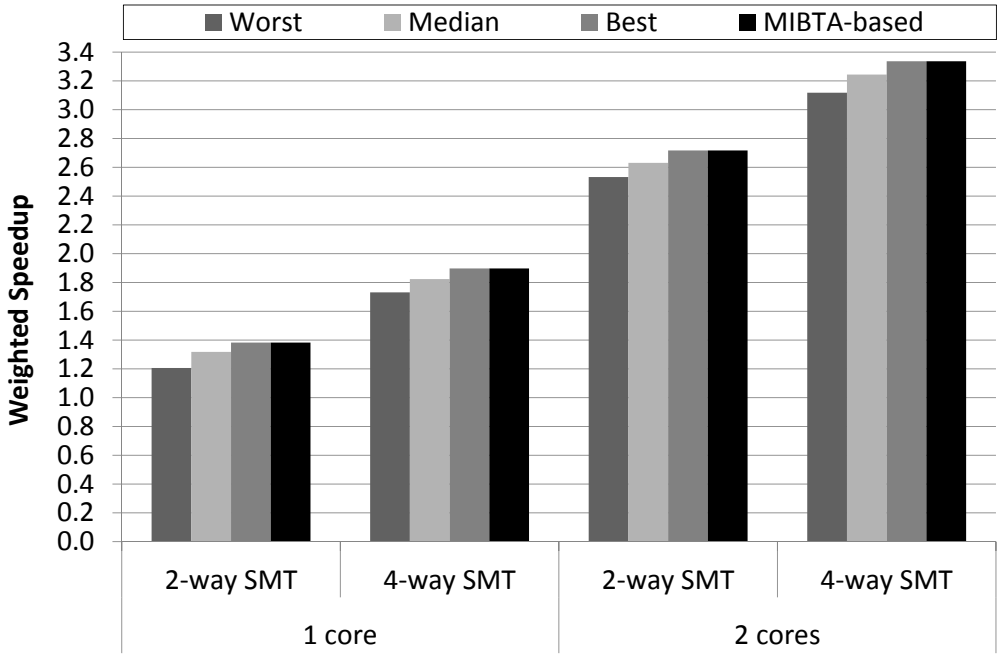


Figure 5.13: Weighted speedup of different symbiotic schedulers for four CMP+SMT configurations

port. ITCA and MIBTA require slightly more hardware support (basically the sampled ATD and the `RSA`, respectively), while CAA presents a nonnegligible hardware cost and complexity as the required hardware blocks are spread throughout all the pipeline of the processor. In contrast, MIBTA relies on isolation phases to predict performance in isolation without introducing expensive hardware support.

5.5 Other Considerations

5.5.1 System-level Considerations

The proposed CPU accounting mechanism for CMP+SMT processors can be applied to several scenarios. First, MIBTA can report the execution time of a task when running alone in the system. This metric reports more accurately the progress of a task than the actual execution time in the CMP+SMT processor. This information can be used by the scheduler to provide fairness, per-task QoS, task prioritization and performance isolation.

To exemplify the potential benefits of MIBTA for the software stack, we perform the following experiment: We combine the feedback provided by MIBTA with the

symbiotic scheduling techniques introduced by Snavely et al. [58, 59]. The SOS scheduler (for Sample, Optimize, Symbios) combines a sample phase which collects information about various possible schedules, and a symbiosis phase which uses that information to predict which schedule will provide the best performance. The sampling phase of the SOS scheduler is very different from the isolation phases of MIBTA: SOS samples some random schedules from the huge amount of possible schedules in a workload. After the sampling phase, SOS predicts the best candidate schedule based on different performance counters measurements. Finally, it runs the candidate optimal schedule for the remaining time.

The original SOS scheduler sought to optimize the weighted speedup of the workload. To that aim, SOS makes use of different heuristics to predict this optimal schedule. Making use of MIBTA, we can *actually measure* the weighted speedup of each random schedule without running each task in isolation in the whole system. MIBTA executes tasks in isolation in each core, and this process is transparent to the OS scheduler. Based on MIBTA's measurements, SOS can select the schedule that reported the best performance in the sampling phase.

Figure 5.13 shows the weighted speedup results with different schedules in four processor configurations with different number of cores (1 and 2), and 2- and 4-way SMT cores. In each configuration, we randomly select a workload with more tasks than available hardware threads. The number of total tasks in the workload is 16 for the 2-core 4-way SMT configuration, and 8 for the remaining ones. SOS generates 20 random schedules (this number is significantly smaller than the total number of available schedules) and decides the candidate optimal schedule for the remaining time. We evaluate the heuristic based on the information provided by MIBTA, and we also report the performance of the worst, median, and best schedule out of the 20 random schedules. We can see that the range of values of the weighted speedup is wide: the best schedule obtains between 7.0% and 14.6% higher weighted speedup than the worst schedule. SOS decisions based on MIBTA's feedback always selects the best performing schedule. This experiment proves that the accurate estimation of MIBTA can be used to improve system-level performance.

Finally, thread-progress aware fetch policies such as the ones presented by Eyerman and Eeckhout [13] could be applied. In this paper, the authors introduce a fetch policy that continuously monitors the progress of each thread in the SMT processor and gives more priority to the threads that are falling behind. Having a more accurate accounting mechanism such as MIBTA would increase the effectiveness of such a proposal.

5.5.2 Virtualized Environments

In data centers, customers are charged according to the use of resources they do. Users are provided with one or several Virtual Machines (VM) in which they can run their tasks. In this case, the CPU capacity is not measured per thread or per task, but per virtual machine: the data center owner charges the user on VM resource-utilization bases.

MIBTA perfectly fits in this type of virtualized environments. The only additional consideration is that, when several virtual machines share the same MT processor, MIBTA has to track the virtual machine to which each task/thread belongs to. Inter-task interferences are no longer considered, but inter-VM interferences. In this case, it would be the hypervisor (virtual machine manager) filling out the thread-task mapping table, setting the same value for all tasks/threads belonging to the same virtual machine.

5.5.3 Dynamic Voltage and Frequency Scaling

Usually DVFS varies the frequency at which cores work keeping the frequency of shared cache and memory unchanged. Consequently, the IPC of a given task may vary with different frequencies. For instance, if a task is memory bound and we decrease core frequency, the IPC of the task will increase since the number of cycles waiting for memory are reduced (measured in the decreased processor frequency). The only change MIBTA requires in the presence of DVFS is a synchronization of the isolation periods of MIBTA with the points in time in which DVFS is changed. Given that changing from one given voltage and frequency operating point to a different one takes in the order of microseconds [23], the overhead of the extra isolation periods of MIBTA will be very low. If required, per-DVFS operating point IPC values of each task in isolation can be maintained either at the hardware level or at the software level.

5.5.4 Parallel Tasks

MIBTA also works for multi-threaded workloads with minimal changes with respect to its implementation shown in previous sections. The interaction between threads in a parallel task can be *positive* when, for example, one thread prefetches data for another thread. This behaviour is intrinsic to the task, and hence it also occurs when running in isolation. Consequently, MIBTA does not need to track it. When a parallel task runs with other tasks, it may suffer *negative* interaction, i.e. it may suffer intertask

contention.

In most of the cases, tasks are bound to some cores: in order to benefit from data sharing, all the threads of a task are usually located onto the same cores. So, a parallel task does not usually share its cores with other tasks. Under this scenario, the parallel task is already running in isolation in the core (not in the LLC), which means that the isolation phases will not degrade the system performance. In the less common case in which several parallel tasks share different cores, MIBTA would have to stop temporally the threads of different tasks and only run threads of the same task in the core during the isolation phase. Given that there are several threads of the same task per core this would reduce the number of isolation phases per core: one per task rather than one per hardware thread as it was the case for multiprogrammed workloads. Consequently, throughput degradation is also reduced. As threads of the parallel task are spread among different cores, MIBTA has to synchronize the different isolation phases in all cores used by the parallel task.

To track intertask LLC conflicts, MIBTA would require an identifier of the parallel task instead of the physical hardware thread. Conflicts between threads of the same task are intrinsic intratask conflicts, and do not have to be tracked by MIBTA. This can easily be done with a hardware table that we denote *thread-task mapping table*. This table has one entry per hardware thread. Each entry contains an integer that ranges from 0 to N-1, where N is the total number of hardware threads in the processor (i.e. number of cores times number of hardware threads per core). All threads of the same task have to be assigned the same value in this table. In the case we have one independent task per hardware thread, each entry in the thread-task mapping table will store a different value.

For instance, if we have a 4-core processor in which each core is 4-way SMT, the thread-task mapping table will have 16 entries. If two parallel tasks run at the same time on the chip such that the first uses the first two cores and the second the last two cores, the contents of the thread-task mapping table would be: $0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1$. On the event of an interaction between two threads in the LLC, MIBTA has to obtain their corresponding task identifiers to determine whether the conflict is an intratask or intertask interference. If conflicting threads have different values in the thread-task mapping table, they belong to different tasks, and consequently, they suffer an intertask interference. If the values in the thread-task mapping table are the same, the interference is regarded as an intratask interference. The thread-task mapping table is made writable to the OS so that it specifies which threads belong to the same task.

At hardware level, MIBTA provides the slowdown that each thread of a parallel task suffers due to intertask interferences. Note that the interaction between threads of the same parallel task is considered intrinsic to the task. Whether the slowdown in a thread translates into a slowdown of the task is something to be determined by the OS or the run-time system. Intuitively, in many tasks there are some threads that are the performance bottlenecks. For example, on a synchronization barrier, the threads getting the latest to the barrier are the bottleneck threads. Any slowdown on those threads due to intertask interferences translates into a task slowdown. It is a responsibility of the OS or run-time system to identify tasks and use per-thread slowdown feedback provided by MIBTA to properly compute tasks' slowdowns.

5.5.5 Scalability

In terms of performance degradation and hardware cost, the worst situation for MIBTA is when all tasks that run in each hardware thread are independent. Under this scenario we have shown that with performance degradations between 1.0% and 1.8%, and reduced hardware budget, MIBTA provides better accounting accuracy than previous CPU accounting mechanisms.

The hardware overhead of MIBTA only depends upon the number of hardware threads per core. This overhead is independent on the number of cores. Consequently, MIBTA scales better with the number of cores, rather than with the number of hardware threads per core. However, current architectures do not implement more than eight hardware threads per core due to its significant hardware cost. Since this trend is predicted to hold for the foreseeable future, MIBTA will scale with upcoming multithreaded systems.

Finally, we have seen that the number of isolation periods reduces with the number of threads per task, and hence the system performance degradation. The rise of parallelism in the last years and its increasing importance will facilitate the use of an CPU accounting mechanism such as MIBTA in the near future.

5.6 Summary

This Chapter has demonstrated that the current CPU accounting mechanisms are not as accurate as they should be in CMP+SMT processors. Though these mechanisms enhance the accuracy of CPU accounting in one of the existing TLP paradigms, they do not consider the interaction of several TLP paradigms at the same time; hence they

introduce inaccuracies in the CPU accounting. To solve this problem, we introduce a novel CPU accounting mechanism denoted *Micro-Isolation Based Time Accounting* (MIBTA). MIBTA reduces the off estimation under 5.0% in average on an 8-core 2-way SMT processor, while previous proposals present average off estimations over 21%. In addition, we have developed a new intertask misses tracking mechanism called *randomized sampled ATD* (RSA). RSA decreases the ATD overhead to 960 bytes in a 2 MB LLC cache, while preserving its high accuracy. At the core level, MIBTA does not track every hardware shared resource, reducing its implementation cost to the minimum.

Chapter 6

CPU Capacity-Aware Scheduling in Multi-Core Processors

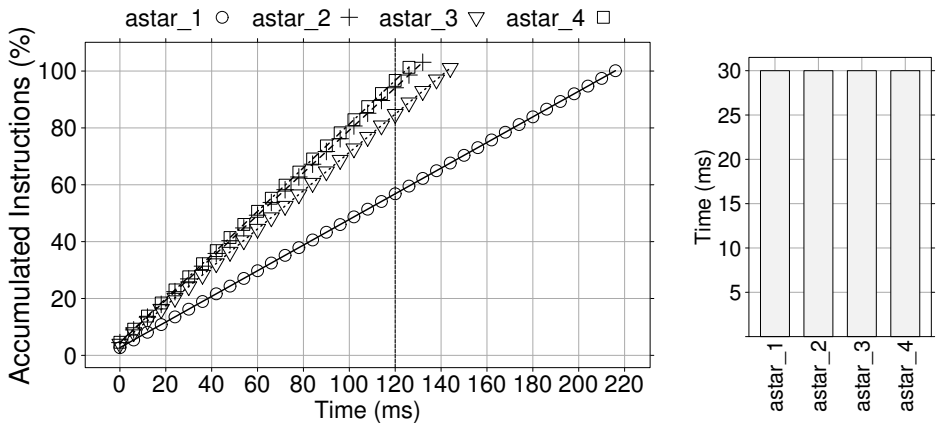
In this Chapter, we analyse how the software level can utilize the CPU accounting. We focus on operating system CPU scheduling algorithm due to impact in computing system performance. We propose for the first time the definition of *effective CPU capacity share* and show how time-based scheduler can be made CPU capacity-aware.

Our results show that by making completely Completely Fair Scheduler (CFS) aware of the CPU capacity enjoyed by each task, we manage to make effective CPU share proportional to the priority of tasks. While performance is roughly unaffected, with an observed difference of less than 0.71% between the time-based and the CPU capacity-based CFS schedulers, CPU capacity-aware CFS improves fairness with respect to time-based CFS by more than 18% in a 4-core configuration.

6.1 Introduction

Tasks running simultaneously on multi-core processors compete for shared resources, such as last-level caches and system resource queues. As a result, in a given amount of running time the actual forward progress accomplished by a process can vary widely depending upon whether the CPU computation capacity is shared and who it is shared with. CPU schedulers, whose goal is to enforce some form of *fair* CPU sharing, have been relying on the running time as the measure of forward progress. While this metric served as a good proxy for progress on single-threaded processors, on multi-core[1] processors it becomes very inaccurate. As a result, the design of schedulers

[1] In fact, the same problem arises in any MT processor. For simplicity in this Chapter, we talk about multi-core processors.



(a) Accumulated executed instructions per each *astar* copy over time (b) CPU time assigned to each *astar* copy after 120ms

Figure 6.1: Executed instructions and CPU time enjoyed by each *astar* copy under CFS in a 4-core processor setup

relying on this metric is fundamentally flawed.

Consider Figure 6.1, which shows the results of an experiment running four copies of the same benchmark (*astar*, SPEC CPU 2006) on a quad-core processor. The four tasks are bound to the same core. On the other cores, we run 12 other tasks (four per core), so that no process balancing is needed. The tasks are managed by Linux CFS, whose goal is to ensure equal progress for processes of equal priority. Figure 6.1 shows the normalized accumulated number of executed instructions over time of the different copies of *astar* (Figure 6.1(a)), and the amount of CPU time each instance of *astar* has received (Figure 6.1(b)). We observe that despite running for the same amount of time on the CPU each copy of *astar* executes a different amount of instructions each time it is scheduled. Consequently, they finish in different times and achieve *unequal amount of forward progress*, and hence received CPU capacity. In this experiment, the slowest copy of *astar* runs 70% slower than the fastest one. Hence, this example clearly shows that despite CFS managing to balance the CPU time (as shown in Figure 6.1(b)), the amount of forward progress made by identical tasks is different. Thus, CFS is unfair in this scenario. Although we used CFS as an example, other time-based CPU schedulers would suffer from the same problem.

Other researchers have characterized the effects of intertask conflicts in the access to shared resources, proposing solutions to alleviate these effects via smarter co-scheduling decisions or modifying the assigned running time to tasks. However, it is still not well understood how to design schedulers that fairly share the multi-

core processor, regardless of how one might want to define fairness. In this work we attempt to close this gap.

In this Chapter, we introduce the concept of *effective CPU capacity share*, which represents the amount of actual CPU *capacity* used by a task when executing. We note that there is more than one way to define effective CPU capacity share, and so we explored two definitions that seemed most intuitive to us. Overall, the contributions of our work can be summarized as follows:

- We quantitatively provide evidence that setting CPU time per task proportional to its priority is not enough to provide fairness in current and future multi-core processors.
- We propose two definitions of effective CPU capacity share applicable to multi-core processors. One definition is based on the notion of task forward progress and another is based on the task utilization of hardware resources which have been covered in Chapter 3.
- Taking CFS as reference time-based CPU scheduler, we make it CPU capacity-aware. We implement our CFS-capacity aware taking into account both definitions of CPU capacity. The followed methodology is not specific for the CFS scheduler, and can be applied to other state-of-the-art time-based schedulers.

The rest of this Chapter is organized as follows. Section 6.2 provides background on CFS, while Section 6.3 introduces our proposed ideal multi-threaded processor and how to map CPU capacity to this baseline. Section 6.4 introduces our novel CPU capacity-aware scheduling algorithms. Section 6.5 evaluates the advantages of these algorithms, and Section 6.6 presents the related work. Finally, Section 6.7 draws the main conclusions of this work.

6.2 Background

In this section, we describe the main aspects of Completely Fair Scheduler (CFS), implemented in recent Linux distributions. We use the CFS as a representative of time-based CPU scheduler.

6.2.1 Completely Fair Scheduler

Linux CFS was introduced in kernel version 2.6.23 to substitute the old O(1) scheduler. The main idea of CFS is to achieve the fairness of an *ideal, precise multi-*

threaded CPU or *ipmCPU* on real hardware. In such a system, all tasks execute simultaneously in parallel receiving an amount of CPU capacity in exact proportion to their weights [31, 45]. If the set of runnable tasks, ϕ, and their weights remain unchanged throughout the interval $[t_1, t_2]$, then each task i will receive $\frac{w_i}{\sum_{j \in \phi} w_j}$ of the total CPU capacity. If all tasks have the same weight, they should receive exactly $1/n$ of the CPU capacity, where *n* is the number of tasks.

In reality, such a model is impractical, because it is not possible to *literally* run multiple tasks in parallel on single-threaded processor. Since the number of tasks that can run simultaneously is limited and tasks cannot be scheduled with infinitesimally small quanta, all practical schedulers try to emulate the *ipmCPU* approximately. To evaluate the fairness of such an approximation, *lag* is the commonly used metric [5], which measures in any given time the difference between the CPU capacity assigned in an *ipmCPU* and the CPU capacity assigned with the current scheduling algorithm. On an *ipmCPU*, unfairness will always be zero, but in a real scene, unfairness will be different to zero. When a task is scheduled on a CPU, it gets 100% of the CPU capacity and decreases its unfairness. In the meantime, all other tasks get 0% of the CPU capacity and increase their unfairness. To model this *ipmCPU*, CFS tracks how fairly each task has been treated relative to the others. Tasks are ordered based on the unfairness and CFS schedules the task with the largest unfairness onto the CPU.

6.2.2 Functioning and Implementation Aspects of CFS

In Linux, a task can be either runnable or blocked waiting for an event to complete, such as an I/O operation or the results of a system call. If a task is runnable, that means that it is competing for the CPU time with the other runnable tasks. Each task belongs to an scheduling class: *Real Time (RT)*, *fair* and *idle* class. The RT class has always priority over other classes and it makes use of an O(1) priority array to manage tasks. The idle class is very special as it is concerned with a task being started when there are no more ready tasks in the system. The fair class implements the CFS and the runnable tasks are stored in the runqueue. Under CFS, it uses a red-black tree[2] for task management, keeping a virtual time line of tasks to schedule: The task with the longest wait time in the red black tree is selected next. With this structure, scheduling decisions are O(1), but task reinsertions are O(log(N)), where N is the number of runnable tasks.

[2] A red-black tree is a data structure representing as a type of self-balancing binary search tree and its operations occur in O(log N) time, where N is the number of nodes in the tree.

To approximate an *ipmCPU*, CFS divides CPU time into epochs and tries to be fair among runnable tasks. At the beginning of the epoch, every task gets a specified time quantum or *timeslice* proportional to its weight. The epoch will finish when all tasks have exhausted their quantum. If the set of runnable tasks and their weights remain unchanged, CFS will be perfectly fair (in terms of time) at the end of each epoch.

To specify the task's priority, CFS makes use of the user-defined `nice` value, similarly to older Linux schedulers. The `nice` value is converted into a task weight that determines the proportion of CPU capacity each task receives. The `nice` value is an integer in the range $[-20, 19]$. A higher nice value corresponds to a smaller priority. Nice values are multiplicative and are translated into weights using an approximation of the following formula: $w = 1024 \cdot 2^{-nice/3.1067}$. Increasing the nice value by 1 implies an increase of 25% in the weight of the task.

As we commented, the CFS assigns CPU time in an epoch proportionally to the weight of each task. Consequently, the CPU time assigned to a task depends upon the load of the system. Equation 6.1 shows the computation of the timeslice assigned to task A in a set of runnable tasks, ϕ.

$$TS_A = \frac{w_A}{\sum_{j \in \phi} w_j} \cdot E \tag{6.1}$$

where w_j is the weight of a task j and E is the epoch length. The denominator is denoted *load of the system*. With this approach, CFS ensures that all runnable tasks receive an execution slot in a given epoch. The epoch length is constant (defined by `sched.latency`), but will be increased if the number of runnable tasks exceeds a threshold.

To track the unfairness evolution of a runnable task, CFS assigns a virtual runtime per task, denoted *vruntime*. This value should be equal for all tasks to ensure fairness. The virtual runtime is a normalized value of the real runtime of a given task with its weight taken into account. Equation 6.2 shows the computation of the virtual runtime of a task A in the instant t.

$$vruntime_A(t) = \frac{w_0}{w_A} \cdot phys\_runtime_A(t) \tag{6.2}$$

where $phys\_runtime_A(t)$ is the cumulative physical runtime of a task A at time t, w_0 is the weight of nice value 0, and w_A is the weight of task A.

CFS sorts tasks in the red-black tree according to the individual virtual runtime

value, and selects the task with the smallest value (the *leftmost* task) to reduce the differences in the virtual runtimes between tasks. As the system progresses forward, executed tasks are put into the tree more and more to the right, slowly but surely giving a chance for every task to become the leftmost task, and thus get on the CPU within a deterministic amount of time.

If the set of runnable tasks and their weights remain constant, the value of the virtual runtimes at the end of the epoch will be exactly the same. For example, if there are three runnable tasks A, B, and C, with `nice` values -2, 0, and 1, respectively, then the corresponding weights are 1586, 1024, and 820. If the epoch length is 20ms, the timeslices will be 9.2ms, 6.0ms, and 4.8ms, respectively. In contrast, the virtual runtime of all tasks will be increased by 6.0ms at the end of the epoch.

To summarize, the virtual runtime only affects the execution order of runnable tasks, while the niceness and the load of the system determine the length of the timeslice of each task in an epoch. If a new task is added to the runqueue, then CFS starts immediately the new one in the current epoch. When a task is no longer runnable, it releases the CPU and is not considered for scheduling until it is runnable again.

6.3 Effective CPU Capacity Share in CMP Processors

So far, we focused our attention on single-threaded processors. Next, we describe the challenges that time-based schedulers encounter in multi-core systems. Then we proceed with the advantages of CPU capacity-aware schedulers.

Figure 6.2 shows a synthetic example with six runnable tasks in the same runqueue in a 4-core processor. Tasks T_A, T_B, T_C, and T_F have the same priority, while tasks T_D and T_E have more priority than the others. CFS assigns T_A, T_B, T_C, and T_F the same timeslice (1 OS tick in this example), while T_D and T_E have longer timeslices (2 and 3 OS ticks, respectively). In this example, the first task to run in the epoch is T_A, and tasks are sorted according to their virtual runtime (see Section 6.2 for details).

Once a task starts running onto a core, it dynamically shares the hardware resource of the multi-core processor with the other tasks running on the other cores, which are selected by the respective scheduler. Hence, each task receives a percentage of the CPU capacity that varies from 0 to 100%. During the timeslice of a task, the co-running tasks in the other cores might change. In Figure 6.2 we show the evolution of the CPU capacity received by each task. Different tasks in the same runqueue can receive significantly different CPU capacity, and this circumstance is out of the control

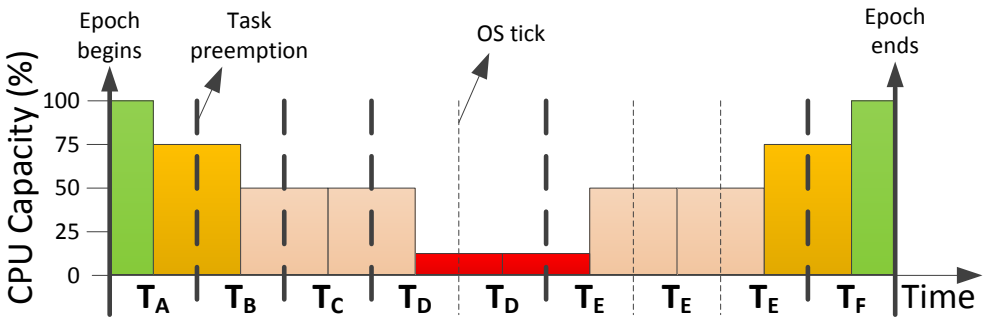


Figure 6.2: Synthetic example with 6 runnable tasks that are scheduled during an epoch and receive different CPU capacity

of the scheduler, directly impacting on the provided fairness. For instance, we observe that once T_A starts its execution temporarily receives 100% of the CPU capacity and after some time, T_A only enjoys 75% of the CPU capacity. This could be because the resource needs of the T_A's co-runners may have changed. We also observe that despite T_A and T_B have the same priority, the former receives higher CPU capacity than the latter on average. Hence, even if the scheduler makes CPU time proportional to task priority, not being aware of the CPU capacity received per task defies its effort in providing fairness.

Our ultimate goal is to design CPU schedulers that provide a fair share of the CPU capacity to each task. This means that all tasks with the same priority must have the same forward progress (T_A, T_B, T_C, and T_F in Figure 6.2). We identify two factors that affect the behavior of CPU capacity-aware schedulers: (1) The definition of CPU capacity itself, and (2) the meaning of a *fair share* of the effective CPU capacity. Both factors have been covered in Chapter 3. In short, the CPU capacity can be defined as *time*, *resources utilization* and *CPU progress*, while fair share of the effective capacity can be defined as *full-share* or *even-share* approaches.

6.4 CPU Capacity-Aware CFS

In this section, we focus on CFS since it is a clear representative of time-based schedulers and it is deployed in real systems. We introduce two variants to CFS to make it CPU capacity-aware, which we call *Balanced CPU Capacity scheduler* (BCCS) and *Equal CPU Capacity scheduler* (ECCS). Our algorithms are compatible with any definition of CPU capacity and fair share of CPU capacity.

Under CFS, each hardware thread (i.e. *core* in a multi-core processor, and *hard-*

ware context in an SMT processor) is perceived as a VCPU. There is a local scheduler per vCPU that works fairly independent from the schedulers in the other vCPUs[3]. Without loss of generality, we follow such an approach, but the same principles apply to systems where CPUs are gang-scheduled [44]. Section 6.5.6 discusses how this approach can be integrated with other schedulers.

6.4.1 Balanced CPU Capacity Scheduler (BCCS)

Similarly to the original CFS, with BCCS each runnable task runs only once in a given epoch and tasks are scheduled based on their virtual runtime. At the beginning of an epoch, BCCS modulates the timeslice of each task based on the CPU capacity received in the past, making it proportional to each task priority.

BCCS involves two main steps to determine the timeslice of each task: First, BCCS checks the CPU capacity that the task received in the past N epochs (N is a parameter that can be defined by the user), and compares this value with the one that it would have obtained in an *ipmCPU*.

In a second step, BCCS modulates the timeslice of each task to match the CPU capacity in an *ipmCPU*. BCCS derives whether a task has received more or less CPU capacity it should, and compensates this situation by adapting the timeslice of each task accordingly.

When a task X starts executing, BCCS uses the timeslice calculated with standard CFS for X. At each epoch boundary, BCCS collects the CPU capacity received in the processor and estimates the CPU capacity of X would have received under the *ipmCPU*. Based on those pieces of information BCCS modifies X's timeslice for the next epoch. At the end of each timeslice, BCCS computes the ratio between these CPU capacities: $CC_X^{ratio} = \frac{CC^{act}}{CC^{ipmCPU}}$. If $CC_X^{ratio} < 1$, X runs slower than in an *ipmCPU* and vice-versa.

Depending on the chosen definition, the CPU capacity received by each task will be:

$$CC_X^{ratio} = \begin{cases} 1, & \text{for time-based schedulers} \\ \frac{RU_X^{act}}{RU_X^{ipmCPU}}, & \text{for resource-based schedulers} \\ \frac{AEI_X^{act}}{AEI_X^{ipmCPU}}, & \text{for progress-based schedulers} \end{cases}$$

where RU_X^{ipmCPU} and RU_X^{act} correspond to the resource usage on an *ipmCPU* and

[3] A *load balancing procedure* might occur if the total load per VCPU is unbalanced, but this is out of the scope of this Thesis.

the current machine, respectively.

Similarly AEI_X^{ipmCPU} and AEI_X^{act} correspond to the accumulate number of executed instructions on an *ipmCPU* and the current machine, respectively. AEI_X^{act} can be obtained by reading performance monitoring counters. To compute AEI_X^{ipmCPU}, BCCS computes the IPC that task X would have had with a fair share of the CPU capacity as $IPC_X^{ipmCPU} = \frac{I_X^{act}}{C_X^{ipmCPU}}$. Hence, the number of instructions that task X would have executed in the *ipmCPU* is given by the following formula:

$$I_X^{ipmCPU} = IPC_X^{ipmCPU} \cdot C_X^{act} \tag{6.3}$$

Hence, BCCS can compute the accumulated executed instructions in an *ipmCPU* by simply adding the number of instructions obtained with Equation 6.3.

BCCS changes the timeslice to compensate this difference. In particular for task i, $\Delta TS_i = TS_i^{act} \cdot (1 - CC_i^{ratio})$, where TS_i^{act} is the current timeslice in the current epoch. BCCS computes the timeslice for the next epoch of a task i as:

$$TS_i^{next} = TS_i^{act} + \Delta TS_i \tag{6.4}$$

Each task requests a given timeslice to reach a fair CPU capacity. Consequently, the requested length of the next epoch is $E^{next} = \sum_{i \in \phi} TS_i^{next}$, and in general, $E \neq E^{next}$.

In order to maintain responsiveness, we decide not to change the default epoch size given by CFS, which in turn may have other side effects in the CFS behavior. Hence, we normalize the timeslice of each task so that E^{next} equals E:

$$TS_{norm,i}^{next} = TS_i^{next} \cdot \frac{E}{E^{next}} \tag{6.5}$$

BCCS balances the CPU capacity of all the tasks. When some tasks go faster than in the *ipmCPU*, while others go slower, BCCS gives more CPU capacity to the slower ones. When all tasks go slower than in the *ipmCPU*, BCCS balances the slowdown each task suffers, meaning that CC_X converges to a value smaller than 1. Analogously, if all tasks go faster than in the *ipmCPU*, BCCS balances the speedup each task experiences, obtaining a CC^{ratio} greater than 1 for all tasks. Note that in an *ipmCPU* or when using the time-based definition of CC^{ratio}, all tasks are receiving a fair share of the CPU capacity and, consequently, BCCS simplifies to CFS ($\Delta TS_i = 0$ in these cases).

6.4.2 Equal CPU Capacity Scheduler (ECCS)

ECCS follows exactly the same steps as BCCS, but follows a different approach to measure the CPU capacity received by each task and the modulation of the timeslice. Again, each runnable task runs only once in a given epoch and tasks are scheduled based on their virtual runtime.

Similarly to BCCS, we first measure the CPU capacity received by each task in the last epoch. Instead of accumulating the values read in the past epochs in the *ipmCPU* and the current machine, ECCS only checks the CPU capacity received in the last epoch by task X:

$$CC_X^{ratio} = \begin{cases} 1, & \text{for time-based schedulers} \\ \frac{RU_X^{act}}{RU_X^{ipmCPU}}, & \text{for resource-based schedulers} \\ C_X^{ipmCPU}/C_X^{act}, & \text{for progress-based schedulers} \end{cases}$$

Then, the CPU capacity received by the task equals $CC_X^{ratio} \cdot TS_X^{act}$. In the next epoch, ECCS seeks to assign a CPU capacity proportional to the priority of each runnable task. ECCS assumes that the interaction among tasks will remain constant. This means that all tasks have to satisfy the following equality:

$$\frac{CC_i^{ratio}}{w_i} \cdot TS_i^{next} = \frac{CC_j^{ratio}}{w_j} \cdot TS_j^{next} \tag{6.6}$$

for all $i, j \in \phi$, where ϕ is the set of runnable tasks and w_i the weight of task i. Equation 6.6 implies that the ratio of CPU capacity assigned to two tasks will be the same as the ratio of their weights: $\frac{CC_i \cdot TS_i^{next}}{CC_j \cdot TS_j^{next}} = \frac{w_i}{w_j}$.

Since we want to keep constant the epoch length, we have to impose this restriction:

$$\sum_{i \in \phi} TS_i^{next} = E \tag{6.7}$$

Combining this restriction (Equation 6.7) with Equation 6.6, we obtain:

$$\begin{aligned} E &= \sum_{j \in \phi} \frac{w_j}{CC_j} \cdot \frac{CC_i}{w_i} \cdot TS_i^{next} \Rightarrow \\ TS_i^{next} &= \frac{w_i}{CC_i} \cdot \frac{E}{\sum_{j \in \phi} \frac{w_j}{CC_j}} \end{aligned} \tag{6.8}$$

ECCS makes use of Equation 6.8 to decide the timeslice of each runnable task. When a task executes for the first time, ECCS assumes a CPU capacity of 1. Note that ECCS using a time-based definition of CPU capacity matches CFS. In fact, ECCS assigns the same timeslices as CFS if tasks are running in an *ipmCPU*, since in this case all execution progresses are equal to 1.

6.4.3 Integrating BCCS and ECCS in CFS

BCCS and ECCS can be easily integrated into CFS since both maintain the same red black tree to sort runnable tasks, the epoch length will be the same as in CFS, and all runnable tasks will execute once in an epoch (check Section 6.4.1 and 6.4.2 for more details). The main changes are focused on the computation of the timeslice and the update of the virtual runtime per task.

BCCS and ECCS also consider the CPU capacity received by each task to modulate the timeslices. Thus, they make use of a different equation to compute the timeslice (Equation 6.5 and 6.8 for BCCS and ECCS, respectively) CFS decides the order in which runnable tasks are executed based on their virtual runtime. To avoid undesired side effects (such as executing a particular task twice in an epoch), we decouple the virtual runtime from the physical runtime:

$$vruntime_A(t) = \frac{TS_A^{CFS}}{TS_A^{act}} \cdot \frac{w_0}{w_A} \cdot phys\_runtime_A(t)$$

In the case that the actual timeslice (TS_A^{act}) matches the timeslice with CFS (TS_A^{CFS}), this equation is the same as in the case of CFS (Equation 6.2). In the case that these values differ, the virtual runtime will be updated with the physical runtime that it would have enjoyed with CFS, giving the illusion to the scheduling algorithm that the timeslice duration is the same as with CFS.

6.5 Evaluation Results

In this section, we perform several studies to evaluate the scheduling algorithms: BCCS, ECCS, and CFS. First, we present the experimental setup used in the case study. Second, we explore the trade-offs in the definitions of CPU capacity and of fair share of CPU capacity for scheduling algorithms. Next, we compare the scheduling algorithms based on time, and ones based on CPU capacity and study the scalability of our CPU capacity-based scheduling algorithms. Finally, we analyse our CPU

capacity-based scheduling algorithms in dynamic behaviour and different priorities.

6.5.1 Experimental Setup

We implement the functionalities of CFS, BCCS, and ECCS in a modified version of the AKULA toolset [69]. AKULA is a scheduler simulator that allows developers to implement and debug scheduling algorithms easily and quickly. In order to generate the data required by AKULA (IPC, execution progress and resource utilization of tasks), we make use of the MPsim simulator studied in Chapter 2 in two processor setups shown in Table 6.1: a 4-core processor with a 4MB L2 cache, and a 16-core processor with a 16MB L2 cache.

Workloads: We feed the MPsim simulator with traces collected from the SPEC CPU 2006 benchmarks (see Chapter 2 for more details). Running all N-task combinations is infeasible as the number of combinations is too high. We select 15 benchmarks to generate 4-task and 16-task workloads and a total 1635 simulations in MPsim are carried out.

We feed our modified AKULA toolset with data obtained by MPsim simulator. In this version of AKULA toolset, associated to each core there is a runqueue, which may have different number of tasks. For instance, in a 4-core processor, we refer to a runqueue setup as $X_1.X_2.X_2.X_4.$, where X_i is the number of tasks in runqueue i. For each runqueue setup, we build a total of 54 workloads in 4-core configuration and a total of 24 workloads in the 16-core configuration. Simulations end when all tasks in the runqueue under study have executed at least one time. Tasks are selected randomly and assigned to each one of the available runqueues so that the number of tasks per runqueue remains balanced. We configure the scheduling algorithms with the following parameters: the initial length of an epoch is 6ms, the minimum granularity is 1ms and hence, the maximum number of tasks per epoch is 6 (with this epoch size). The frequency of an OS tick is 1MHz.

Metrics: In all the experiments, we evaluate the scheduling algorithms based on their throughput ($IPC\ mean_i$), which we measure as the average IPC of all tasks in the runqueue i across all epochs; the execution progress min-max ratio ($min - max\ ratio_i^{EP}$); and the execution progress standard deviation ($std\ deviation_i^{EP}$). The last two metrics are crucial to report the fairness results of these algorithms. The execution progress min-max ratio reports the relative lag between runnable tasks in a runqueue, while $std\ deviation_i^{EP}$ measures the standard deviation in the execution

Table 6.1: MPsim simulator configuration

Number of cores	4 & 16
Fetch bandwidth	8 inst. per cycles
Issue queues sizes	48 int, 48 fp, 48 ld/st
Execution units	4 int, 2 fp, 2 ld/st
Back end	132 int/fp physical registers, 160-entry ROB
Branch predictor	Perceptron 256 global-entry, 40 global-H, 4K local-entry, 14 local-H, 100-entry RAS
Target frequency	2.0GHz
Icache (per core)	64KB, 4 ways, 1 bank, 128B line, 1-cycle access
Dcache (per core)	64KB, 8 ways, 1 bank, 128B line, 1-cycle access
L2 cache (Shared)	4MB, and 16MB, 16 ways, 8 banks, 128B line, 15-cycle access
MSHR	32 entries
Memory latency	300-cycle access

progress of runnable tasks in a runqueue i. The different metrics are defined as follows

$$
\begin{aligned}
IPC\ mean_i &= \frac{\sum_{j\in\phi} AEI_j^{act}}{\sum_{j\in\phi} C_j^{act}} \\
min-max\ ratio_i^{EP} &= \frac{\min_{j\in\phi} EP_j}{\max_{j\in\phi} EP_j} \\
std\ deviation_i^{EP} &= \sqrt{\frac{1}{N-1}\sum_{j=1}^{N}(EP_j - \overline{EP})^2}
\end{aligned}
$$

where ϕ is the set of runnable tasks in a runqueue i, EP_j the execution progress of tasks j and is calculated as $\frac{AEI_j^{act}}{AEI_j^{ipmCPU}}$, N is the number of runnable tasks in the runqueue, and $\overline{EP}$ is the average execution progress in the runqueue.

To measure resource utilization per task, we choose a possible implementation of Equation 3.1: $RU_X^{act} = RU_X^{LLC} = LLC_{occup,X}$, where $LLC_{occup,X}$ is the percentage of LLC cache blocks owned by task X[4]. This formula considers as an appropriate metric of resource utilization, how many shared resources are being used (LLC cache is the most important one in our configuration) and for how long. In the case of the *ipmCPU*, $RU_X^{ipmCPU} = LLC_{occup,X}^{ipmCPU}$, where $LLC_{occup,X}^{ipmCPU}$ might be the entire LLC or simply 1/N-th.

[4]We are assuming single-threaded tasks, in Section 6.5.6 we show that multi-threaded tasks can also be easily managed with our approach.

6.5.2 CPU Capacity-based Schedulers Self-Evaluation

As explained in Chapter 3, we introduced two novel definitions of CPU capacity (based on *resource utilization* and *execution progress*) and two possible definitions of the fair share of the CPU capacity (*even-share* or *full-share* of the resources). Consequently, four possible flavours of each algorithm can be evaluated. In this section, we focus on a single algorithm (ECCS) to explore the different trade-offs in these four baselines. The same conclusions apply to BCCS, but we omitted the figures for the sake of clarity. In order to avoid uncontrolled side-effects, we assume for all setups that the required hardware support to track execution progress or resource utilization already exists, and that it has a perfect accuracy. In the next section, we will explore the effect of using non-ideal, but implementable hardware support.

Figure 6.3 shows the IPC mean, min-max ratio and standard deviation of the progress, and the LLC occupancy of the tasks running in the first runqueue for a 4-core setup when using progress and resource utilization as CPU capacity (`P` or `R` in the caption); and an even-share or full-share as the *fair* share of CPU capacity (`even` or `full` in the caption). For example, `ECCS_P_even` stands for ECCS based on progress with an even share of the resources.

Let us start comparing the effect of using progress-based CPU capacity versus resource-based CPU capacity. In terms of IPC mean the results are similar, with resource-based schedulers obtaining slightly better results (8.4% average improvements). However, this improvement in IPC mean comes at a significant cost in fairness. In terms of progress min-max ratio, progress-based schedulers significantly improve over resource-based ones, nearly reaching the optimal value of 1. In contrast, resource-based schedulers have values ranging from 0.25 to 0.54 (56.3% average reduction in fairness). The same trend is observed in terms of standard deviation of execution progress, which is reduced from 0.39 for resource-based schedulers to less than 0.01 for progress-based schedulers on average.

Resource-based schedulers increase the timeslices of tasks with low LLC occupancy. In general, these tasks have higher IPC values, increasing average IPC. In contrast, LLC cache hungry tasks are penalized for their usage of the LLC, independently of whether they are using this space or thrashing other tasks in the system. This means that resource-based schedulers increase the progress of CPU-bound tasks, but reduce the progress of cache-sensitive and cache-thrashing tasks. This is the reason why resource-based schedulers obtain better IPC mean than progress-based schedulers. Next, we observe that resource utilization per task is more balanced for

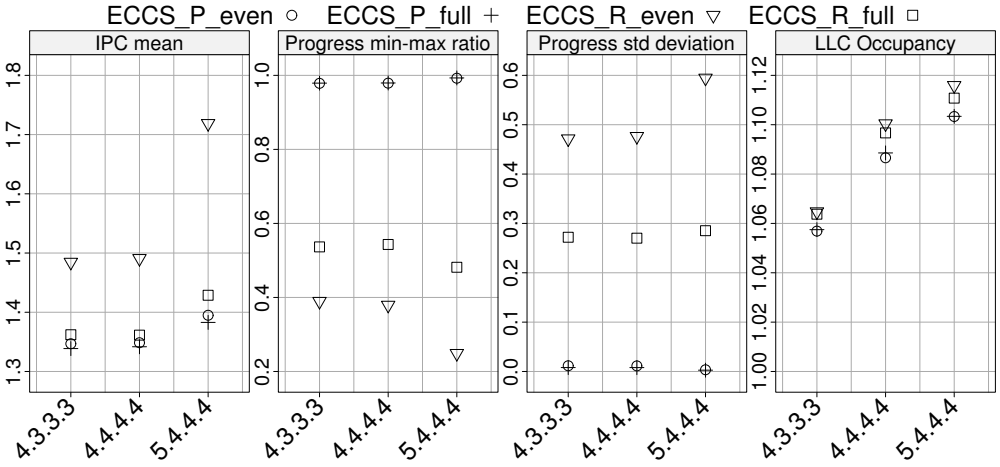


Figure 6.3: IPC mean, fairness, and LLC occupancy (measured in MB) results with a 4-core multi-core processor for the tasks running in the first runqueue. Configuration $X.Y.Z.W$ indicates the number of tasks running on each runqueue (with a total of 13, 16 and 17 tasks in the system). Two definitions of CPU capacity and fair share of CPU capacity under ECCS algorithm are used

resource-based schedulers (which is the objective of the algorithm), but LLC occupancy per task does not significantly increase (less than 1% variations). LLC occupancy mainly depends on the co-scheduled tasks on the multi-core processor for the evaluated timeslice lengths. Note that more sophisticated formulas to compute RU_X^{act} can be developed, and that they might consider the time and the amount of resources enjoyed by the task, but also the utility of these resources for the task. Designing more appropriate formulas for Equation 3.1 is part of our future work.

When we consider using an even-share or a full-share of the CPU capacity, we appreciate no noticeable difference among them under all the evaluated metrics. Note that in general, tasks running on a multi-core processor will go faster than running with only an even share of the hardware resources and, consequently, the execution progress will be larger than one. In contrast, tasks usually run slower than with the full share of the hardware resources. ECCS covers both situations with similar results: `ECCS_even` obtains slightly better results in IPC mean, but slightly worse in fairness results.

To conclude, progress-based schedulers clearly outperform resource-based schedulers in terms of fairness without significantly affecting IPC mean. The best fairness results are obtained with the full-share approach to measure the fair share of CPU capacity (although very close to results with the even-share approach). For these reasons and since there are several hardware proposals in the literature that provide the

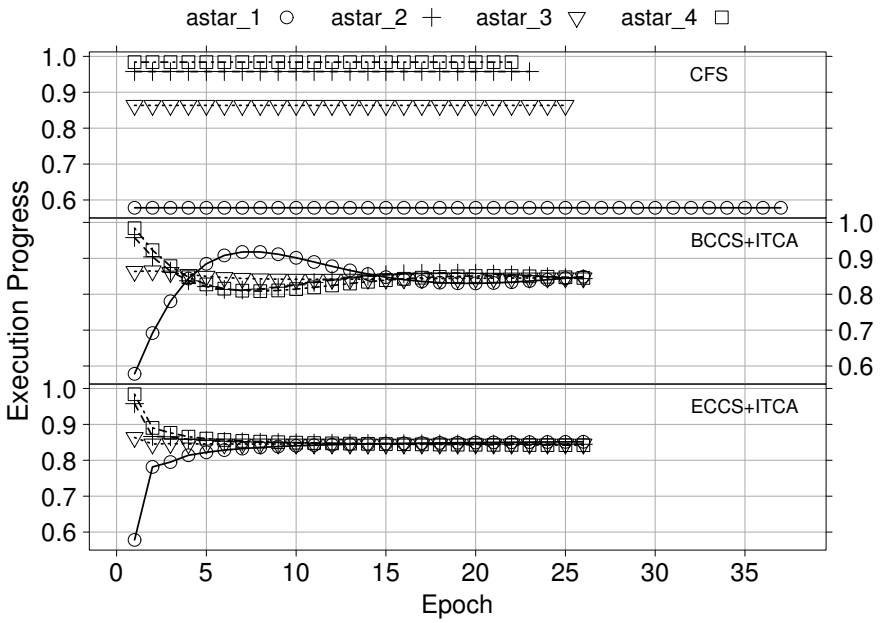


Figure 6.4: Progress evolution of each `astar` copy running on the same runqueue in a 4-core multi-core processor. The same experiment is performed using CFS, `BCCS+ITCA`, and `ECCS+ITCA` algorithms

functionality required by full-share approach studied in Chapters 4 and 5 [33, 13, 35], we make use of our algorithms with the progress and the full-share approach in the remaining sections of this Chapter.

6.5.3 Time-based vs. CPU Capacity-based Schedulers

Our next experiment seeks to prove the ability of our proposed CPU capacity-based scheduling algorithms to balance execution progress among runnable tasks. Figure 6.4 shows the execution progress of four copies of the benchmark *astar* running in the same runqueue. In this experiment, we have four cores, with four tasks running on each core (the tasks in the other runqueues are randomly selected, and are not shown for the sake of clarity), all with the same priority and with implementable hardware support (ITCA [33]). As illustrated in the introduction (see Figure 6.1), CFS is not aware of the CPU capacity each *astar* copy receives, which in this case ranges from 0.58 to 0.98 depending upon the co-runners in the chip. As a result, each copy of *astar* ends executing in very different times (up to 70% difference).

In contrast, `BCCS+ITCA` and `ECCS+ITCA` take into account the execution progress of each copy of *astar* to modulate the timeslice of each task. As a result, both

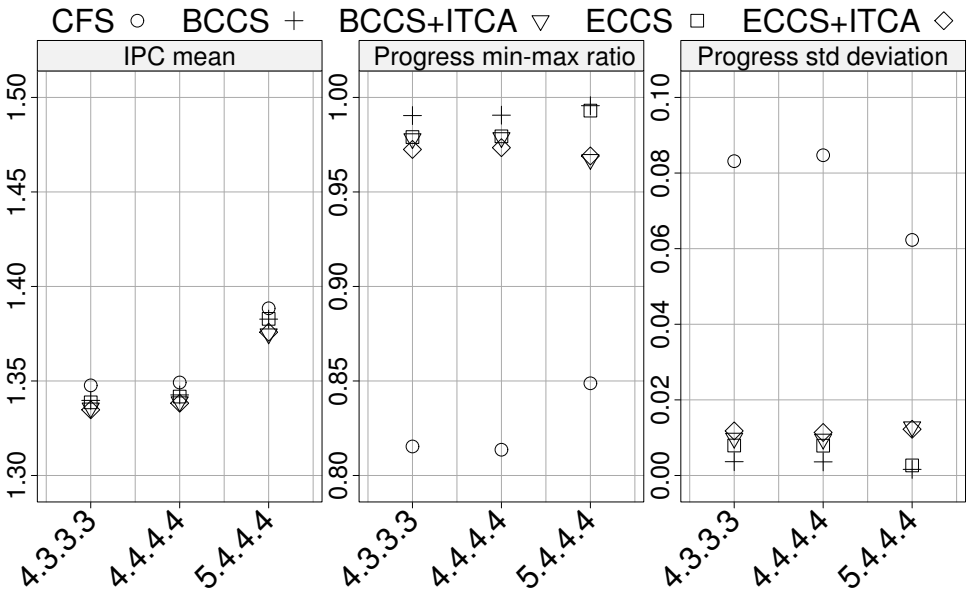


Figure 6.5: IPC mean and fairness results with a 4-core multi-core processor for the tasks running in the first runqueue. Configuration $X.Y.Z.W$ indicates the number of tasks running on each runqueue (with a total of 13, 16 and 17 tasks in the system)

approaches manage to make all copies of *astar* finish in the same epoch (number 26). Since `BCCS+ITCA` takes into account the CPU capacity received in all previous epochs, it tends to overreact to really unbalanced situations (as at the end of the first epoch), and the execution progress of *astar_1* oscillates around the final value of 0.85. In contrast, `ECCS+ITCA` converges faster to this value as it only takes into account the execution progress in the last epoch.

Next, we evaluate the IPC mean and fairness results of these three schedulers with a whole set of experiments. We evaluate a 4-core setup, with a balanced number of tasks per runqueue (three, four or five). Figure 6.5 summarizes the results for the tasks running in the first runqueue. We observe that all schedulers obtain the same IPC mean (less than 0.71% average differences). CFS presents much worse fairness results than `BCCS+ITCA` and `ECCS+ITCA`. On average, the min-max ratio for CFS is 0.83, while `BCCS+ITCA` and `ECCS+ITCA` reach 0.975 and 0.972, respectively (up to 18% increase in fairness). The same happens with the standard deviation of the execution progress, which is reduced from 0.077 with CFS, to only 0.011 with `BCCS+ITCA` and 0.012 with `ECCS+ITCA`.

We also evaluate our algorithms with an ideal hardware support, neglecting the ef-

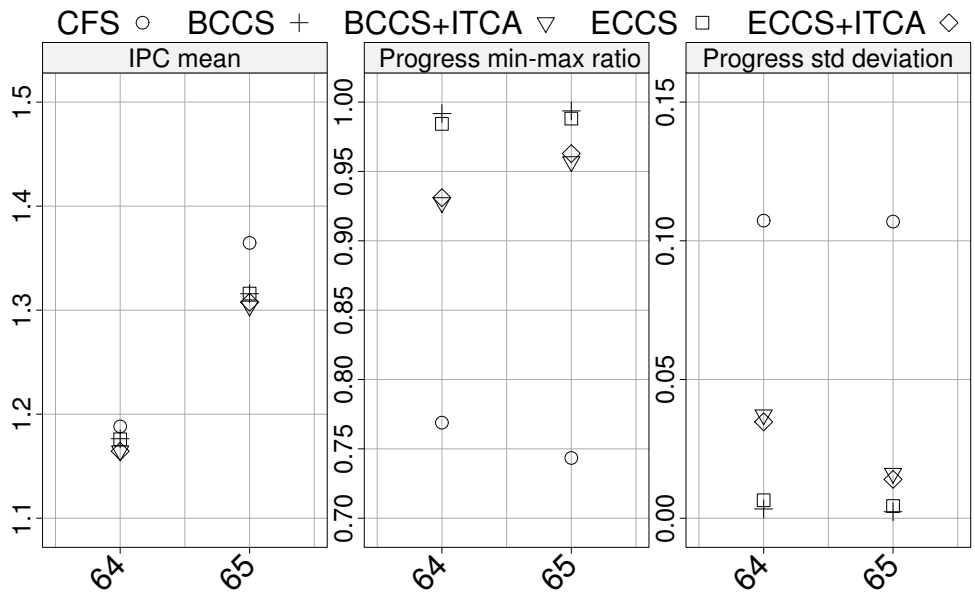


Figure 6.6: IPC mean and fairness results with a 16-core multi-core processor for the tasks running in the first runqueue. In the first runqueue a total of four or five tasks are executing, while four tasks are in the remaining 15 runqueues (with a total of 64 and 65 tasks in the system)

fects introduced by ITCA inaccuracies (BCCS and ECCS in Figure 6.5). In this case, the IPC mean results are basically the same as with the version based on ITCA, while the fairness results slightly improve: 0.99 for the min-max ratio and less than 0.01 for the standard deviation of execution progress. These results indicate that (1) our algorithms are fair in terms of CPU capacity and (2) current hardware support (ITCA) provides accurate CPU capacity estimations, since near optimal fairness results are obtained when ITCA is deployed. Finally, BCCS obtains slightly better fairness results than ECCS in the long term in stable scenarios, but converges more slowly to the optimal solution.

6.5.4 Scalability Analysis to Large Core Counts

In this section, we perform a similar experiment with a multi-core processor with a larger number of cores. Figure 6.6 shows the IPC mean and fairness results obtained in a configuration with 16 cores for the tasks running in the first runqueue. In this experiment, four or five tasks are running in the first runqueue, while four tasks are running in the remaining 15 runqueues (64 and 65 tasks in total).

The results presented in Figure 6.6 show a similar trend to the ones presented in the previous section (check Figure 6.5). We observe that all the evaluated algorithms achieve a very similar IPC mean (CFS obtains 2.8% higher IPC mean), and that the ideal hardware support for CPU capacity accounting does not improve this metric. In contrast, `BCCS+ITCA` and `ECCS+ITCA` attain much better results than CFS in progres min-max ratio. On average, the min-max ratio for CFS is 0.76, while our scheduling proposals obtain 0.95 (an average 25% improvement).

In terms of the execution progress standard deviation, our scheduling proposals obtain similar results to the ones presented in the previous section. However, the difference between CFS, and `BCCS+ITCA` and `ECCS+ITCA` is larger in the 16-core system (an average reduction from 0.11 to 0.026). Since the number of running tasks in the system increases, a more heterogeneous resource usage occurs, harming the fairness results of CFS.

In contrast, the schedulers BCCS and ECCS obtain the same IPC mean results than `BCCS+ITCA` and `ECCS+ITCA`, while they reach better results in both fairness metrics, nearly reaching the ideal results. The accuracy of ITCA degrades with the number of cores in the system, since network-on-chip and memory bandwidth contention is partially taken into account in ITCA, and these resources become more contended in larger systems. Thus, we conclude that our scheduling proposals can adapt to systems with a large number of cores and distribute a fair share of CPU capacity to all runnable tasks without affecting the performance in the system.

6.5.5 Case Studies

6.5.5.1 Dynamic Behaviour

In the previous experiments, the execution progress of each task depends only on the co-runners on-chip. To model a more dynamic scenario, we perform an experiment in which the execution progress of a task degrades by 10% after two executions. This experiment is intended to illustrate the adaptation capacity of the different algorithms. We finalize the simulation when all tasks in the first runqueue have executed four times. In this section we focus on `ECCS+ITCA` only because the results with `BCCS+ITCA` present similar trends.

Figure 6.7 shows the evolution of the execution progress of four copies of `astar` running in the same runqueue in a 4-core processor setup. The execution progress of a task X is calculated as $\frac{I_X^{act}}{I_X^{ipmCPU}}$ in the case study. In the case of CFS, *astar_1* progresses much slower than the other tasks, lagging behind them. When *astar_1*

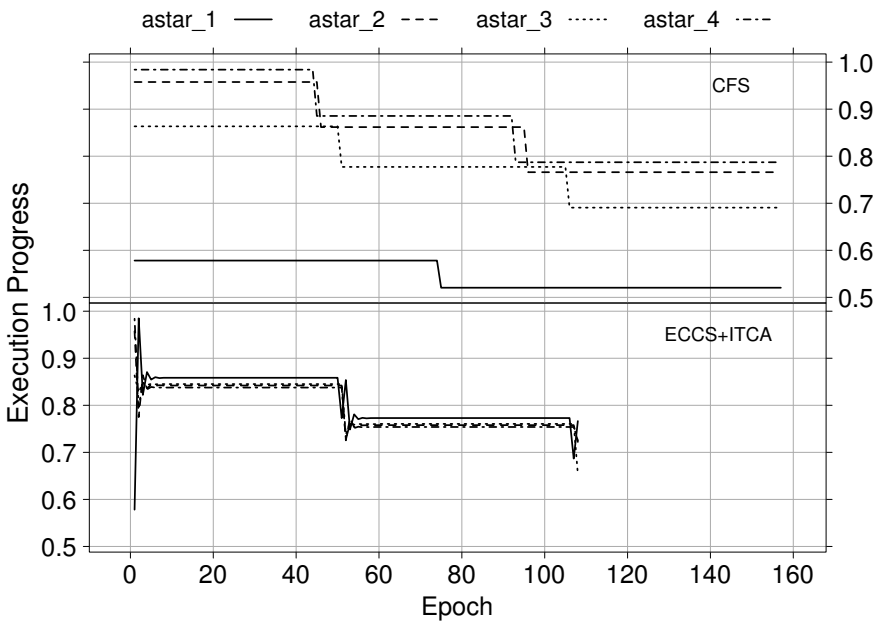


Figure 6.7: Progress evolution of each `astar` copy running on the same runqueue in a 4-core multi-core processor. After two executions, the execution progress of the `astar` copy degrades by 10%. The same experiment is performed using CFS and `ECCS+ITCA` algorithms

finalizes executing for the fourth time (epoch 156), *astar_2* and *astar_4* already executed 6 times. In contrast, `ECCS+ITCA` manages to balance the execution progress of all tasks even when this progress degrades after 2 executions (all tasks have executed four times after 108 epochs). Furthermore, `ECCS+ITCA` algorithm quickly converges to a balanced progress for all tasks.

6.5.5.2 Different Priorities

Next, we perform a case study that illustrates the effectiveness of `ECCS+ITCA` in a scenario with tasks with different priorities. As explained in Section 6.4, this algorithm considers the user-defined priorities when computing the timeslice of each runnable task.

Figure 6.8 shows the number of executed instructions of four copies of `astar` with different priorities. The first two copies (`astar_1` and `astar_2`) have a `nice` value of 2, while the last two copies (`astar_3` and `astar_4`) have a `nice` value of 0. In this scenario, the ratio between the weights of the first two copies and the last two is 1.56. In an ideal, precise multi-threaded CPU, the second pair should execute 56% more instructions than the first one. However, we observe that under CFS, `astar_4`

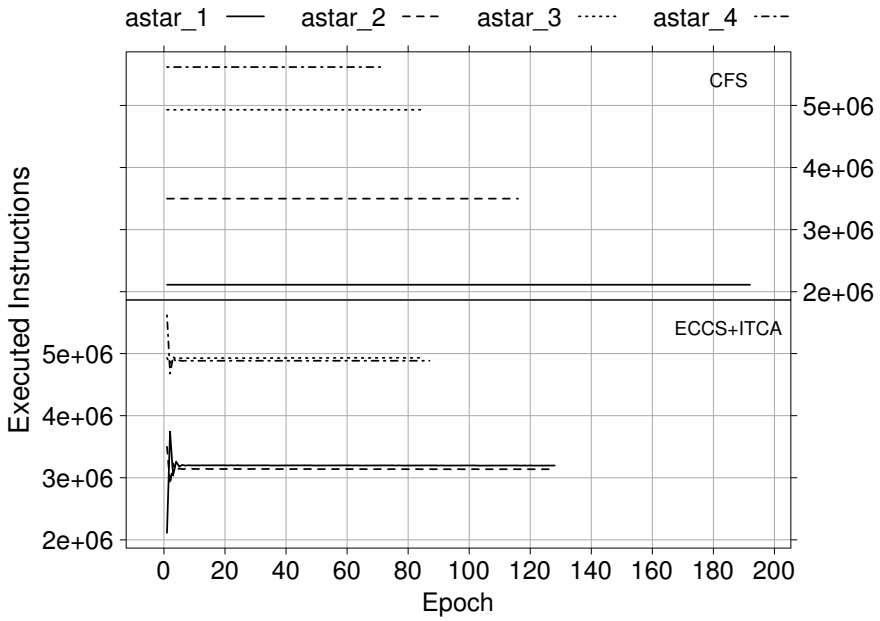


Figure 6.8: Executed instructions evolution of each `astar` copy running on the same run-queue in a 4-core multi-core processor with different priorities. The same experiment is performed using CFS and `ECCS+ITCA` algorithms

is executing 2.7 times more instructions than `astar_1`. As shown in Figure 6.1, each copy of `astar` has a significantly different execution progress in this workload, but CFS is unaware of this situation.

In contrast, `ECCS+ITCA` manages to balance the number of executed instructions of each copy with the same `nice` value, while the ratio of executed instructions of copies with different weights is 1.58, very close to the goal of 1.56 determined by the weights of the tasks. Note, that `ECCS+ITCA` reaches the right timeslice values after just 2 epochs of exploration.

6.5.6 Discussion

Our focus in this Chapter has been on multi-core processors. However, we note that the same problems detected for time-based schedulers in multi-core processors apply to other architectures with on-chip shared resources such as SMT processors, fine-grain multi-threaded processors, or any combination of them and multi-core processors. Our scheduling algorithms are orthogonal to the underlying processor architecture as long as the appropriate CPU accounting mechanism (hardware support) is in place to measure CPU capacity. Such hardware support has been proposed in the

literature for SMT [13], multi-core [33], and hybrid CMP+SMT processors [35] (i.e. each core is an SMT processor) studied in Chapters 4 and 5. In fact the latter is the design followed by several major chip vendors such as the IBM POWER7 [56] or the Intel Core i7 [51].

While the current Chapter presented the application of CPU capacity-aware scheduler to CFS, the technique is general and can be applied to other schedulers as well, since most of them use CPU time as the proxy for progress. For instance, Solaris time-sharing scheduler adjusts the timeslices of tasks based on floating priority values. A priority value depends on whether the task has used up its previous timeslice or gave up the CPU before its expiration. To integrate capacity-aware scheduling into Solaris, we could redefine its algorithm for adjusting priorities. For example, the priority value would depend not only on whether the task has used up its previous timeslice, but also on how much effective CPU capacity it received while it ran. The same principles apply to other schedulers based on *multilevel feedback queues* [10] that many commodity OS's implement and rely on CPU time to measure progress.

Although not shown in the evaluation of this Chapter, parallel tasks are treated similarly to single-threaded tasks under these schedulers. The only difference is that the CPU capacity used by a parallel task corresponds to the total CPU capacity used by each of its threads. The hardware proposals to measure CPU capacity already deal with multi-threaded tasks identifying the CPU capacity consumed by each threads [33, 35, 13]. Thus, BCCS and ECCS can work with parallel tasks.

6.6 Related Work

Improving scheduling algorithms for current and future multi-core systems has been extensively studied both at software and hardware level. In this section, we briefly describe recent work that is relevant to our proposal.

To avoid resource contention, several approaches propose models that predict the impact of interferences among co-running tasks to system performance. Snavely et al. [58, 59] present the scheduler *Sample, Optimize, Symbios* (SOS), which combines a sample phase to collect information about a reduced set of schedules, and a symbiosis phase that makes use of this information to predict the best performing schedule. This proposal has two drawbacks: in the sampling phase, the workload is not scheduled optimally and the sampling phase increases both number of threads and of cores. Eyerman and Eeckhout [14] propose a probabilistic job symbiosis model that enhances the SOS scheduler. Based on the cycle accounting architecture [13], the model es-

timates the single-threaded progress for each job in a multiprogrammed workload, improving the accuracy of the prediction mechanism. Finally, Settle et al. [53] predict cache contention in an SMT processor based on off-line profiles of applications' cache activity, and select schedules with tasks that are predicted to have less cache conflicts. All these proposals seek to find a *good* schedule, but once the schedule is decided, they rely on time-based metrics to decide timeslices and no tracking of forward progress is performed, resulting in less fair systems.

Fedorova et al. [16] propose a software solution which is based on the concept of compensation. Whenever the OS detects that a task does not make the progress it is supposed to make, the OS increases the time quantum of the task, giving more temporal resources to the task and, thus, allowing the task to reach its expected performance. While Fedorova's solution has similar underlying principles, its key shortcomings are as follows. It implies only a single definition of CPU capacity, one based on equal resource distribution. Our current solution, on the other hand, can work with different definitions of CPU capacity. Further, Fedorova's solution relies on an analytical processor model, which was tractable for the simple Niagara processor that Fedorova's work targeted, but would be unsuitable for more complex super-scalar processors addressed in this work.

Other approaches [11, 29, 68, 26] propose to classify tasks in term of their cache miss rates. These approaches determine whether there is contention by examining tasks' miss rates and scheduling decisions that lead to high miss rates are avoided. However, relying on miss rates is imprecise, because they are not the most appropriate predictors of contention, specially in SMT processors.

Ebrahimi et al [12] propose a hardware mechanism (*Fairness via Source Throttling* (FST)) to enable fairness in the entire memory system. FST estimates the unfairness per task in each interval and dynamically adjusts the memory request rate per task, based on a user-defined unfairness threshold. FST can be used in conjunction with the scheduler to provide different fairness metrics such as minimum slowdown per task.

Solaris recently extended its load-balancing mechanism so that a task's notion of utilization (or required resources) is proportional to its scheduling priority [47]. This allows the scheduler to load-balance lower-priority tasks away from where high-priority tasks are running, automatically reducing contention for resources. Although Solaris has some notion of the resource usage, it is not aware of the execution progress of tasks, and does not differentiate between tasks with the same priority. Finally, this optimization requires that there are some spare logical CPUs so that high priority tasks

receive more resources.

More recently, Van Craeynest et al. [65] present fairness-aware scheduling techniques for single-ISA heterogeneous multi-core processors with big and small cores. The proposed technique determines for how long each task has to run in each core type, so that all of them suffer the same slowdown with respect to execution in isolation on the big core. Since the performance of each task reacts differently to having more or less resources, the scheduled time of each task on the two core types will be different to equalize its slowdown.

Although we share the same objective of increasing system fairness, there are fundamental differences with our approach. First, we focus on homogeneous architectures with a single core type, where it is infeasible to balance task progress by pinning them to different cores. Thus, the proposals of Van Craeynest et al. [65] would not report any benefit over current time-based schedulers in homogeneous architectures. Second, they seek to be fair among the tasks simultaneously running for a period of time, while we want to be fair among all the tasks that are time-sharing a given core. Third, they work with a simplified scheduler that would be difficult to combine with advanced scheduling techniques such as task priorities or load balancing mechanisms. In contrast, we focus on a world-wide commercially used scheduler such as the Linux CFS, showing how it can be adapted to make it CPU capacity aware. Finally, Van Craeynest et al. [65] do not consider conflicts in shared hardware resources for scheduling decisions. Nevertheless, combining both approaches could be very interesting to be fair among all the tasks running in future heterogeneous systems.

6.7 Summary

In this Chapter, we have provided evidences that time-based CPU schedulers manage to balance CPU time among running tasks in multi-core processors, but they do not balance tasks' forward progress. Schedulers relying on this metric are fundamentally flawed and, as a consequence, are not providing fairness in multi-core systems. We have introduced the concept of *effective CPU capacity share* in multi-core processors and can be mapped in several suitable definitions studied in Chapter 3. We have developed several CPU scheduling algorithms that consider CPU capacity in their scheduling decisions. Our results show that using tasks execution progress as a measure of CPU capacity is better than using the resource utilization incurred by each task. We appreciate no difference in assuming as *fair share of CPU capacity* either an even-share or a full-share approach.

Taking Linux CFS as a representative of current time-based schedulers, we have shown that our schedulers provide very similar performance to CFS while achieving much more fairness for all tasks (an average 25% improvement in a 16-core configuration). Finally, this approach can be integrated with other state-of-the-art schedulers, not just CFS, since most of them make use of time as the proxy for execution progress.

Chapter 7
Conclusions

In recent years, multi-threaded processors have become widely used in academia and industry in order to increase the system aggregated performance and per-task performance. Multi-threaded architectures increase hardware resource utilization, while reducing design costs and average power consumption by exploiting design re-use and simpler processor cores. Currently, multi-threaded processors are the mainstream in the processor design and are widely used in servers, desktop computers, lap-tops, and mobile devices.

On the other hand, in multi-threaded processors, the performance of a task depends upon both the time the task runs on to the processor and the amount of resources it receives during that time. The latter is in general not under the control of the user or the operating system. To make things worse, there is a non-linear relation between the percentage of resources assigned to a task and the slowdown it suffers with respect to running in isolation with all resources. This situation introduces complexities in the accounting of CPU capacity.

This Thesis presents several proposals that effectively track the CPU capacity to account each task running onto MT processors. The contributions are divided into three groups. First, we formally define the CPU capacity and CPU accounting for MT processors. Second, at the hardware level, we propose two novel CPU accounting mechanisms that improve the accuracy in measuring the CPU capacity in MT processors. To start with, we study CPU accounting mechanisms for CMP processors. Next, CPU accounting mechanisms are studied for CMP+SMT processors, where each core supports SMT. Finally, at the software level, we study how the OS CPU scheduler should be adapted to take advantage of the proposed CPU accounting hardware mechanisms to increase system fairness. The following sections briefly summarize each one of the contributions.

7.1 Thesis Contributions

7.1.1 Concept of CPU Capacity and CPU Accounting for Multi-Threaded Processors

ST uniprocessor systems use the time a task is scheduled on a VCPU as proxy metric for measuring CPU capacity of tasks. This metric is not valid in a MT processor due to shared hardware resources. In other words, the CPU capacity of a task can be different depending upon co-runner tasks in the processor. For this reason, we have provided two definitions of CPU capacity for MT processors: (*resource utilization* and *CPU progress*) that cover a wide range of potentially interesting scenarios. We have also defined the concept of *CPU accounting* in order measure the CPU capacity of a task running onto a VCPU as if the task is alone running onto a processor. Moreover, we have introduced the *Principle of Accounting*, which specifies that the CPU capacity of a task is roughly the same regardless of the workload in which it is executed.

We have also studied two definitions of fair share of CPU capacity: *full-share* and *even-share* approach. As we focus on CPU capacity as CPU progress, the full-share approach (full-share accounting) considers that the progress it should be accounted is this progress it would take this task to run in isolation. In even-share approach (even-share accounting), it considers that a task should be accounted for the progress it takes the task to finish its execution with an even part of the processor resources.

7.1.2 CPU Accounting for Multi-Core Processors

The off-core hardware resources in a CMP processor are shared among cores. As a result, the utilization of these resources are improved because the resources a task does not require, can be utilized by other tasks running simultaneously onto the processor. For this reason, the system throughput is improved as a result.

In this Thesis, we have shown that the contention of off-core hardware resources affects the accuracy of current CPU accounting mechanisms. Moreover, we have proposed the first CPU accounting mechanism for CMP processors that is aware of interference in shared hardware resources, denoted *InterTask Conflict-Aware* (ITCA) accounting. ITCA recovers the Principle of Accounting in CMP processors. We have evaluated ITCA in several CMP configurations with two, four, and eight cores. The presented results show that the ITCA mechanism significantly improves the accuracy of current CPU accounting mechanisms. In addition, we have improved the accuracy of the ITCA mechanism, denoted I^2TCA, without increasing its implementation com-

plexity. Our CPU accounting mechanism works in different accounting reference: full-share and even-share approaches.

Finally, we have shown that an accurate CPU accounting mechanism is still needed in processors that implement dynamic cache partitioning schemes. Our CPU accounting mechanism adapts to these processors without increasing the hardware overhead, since they can reuse the hardware support required for the cache partitioning schemes.

7.1.3 CPU Accounting for CMP+SMT Processors

The combination of different TLP paradigms is used by processor vendors in order to combine the benefits of each TLP paradigm. As a result, throughput can be improved without increasing the power consumption.

In this Thesis, we have proved that the current CPU accounting mechanisms enhanced the accuracy of CPU accounting in one of the existing TLP paradigms. However, these mechanisms introduce inaccuracies in the CPU accounting for CMP+SMT processors because they do not consider the interference of several TLP paradigms at the same time. For this reason, we have proposed a new CPU accounting mechanism, called *Micro-Isolation Based Time Accounting* (MIBTA), that covers this gap.

MIBTA makes use of a time sampling technique in which tasks run in isolation for short periods of time, with negligible effect on the system throughput and with minimal hardware overhead. We have evaluated our proposal in several CMP+SMT processor configurations and the results show that MIBTA provides an accurate CPU accounting without degrading overall system performance.

7.1.4 CPU Capacity-Aware Scheduling in Multi-Core Processors

Modern time-sharing CPU schedulers multiplex the CPU time for the different tasks. The CPU scheduler is in charge of this operation and selects the best task to run. The CPU scheduler does not directly use the CPU accounting for this decision, but uses it indirectly, considering that the progress of a task is measured in terms of the time spent on the CPU. The CPU scheduler is assuming that the task makes use of the 100% of the processor's hardware resources. With this, the CPU scheduler tries to be fair sharing the CPU time between the tasks in the system. For MT processors, which share a large amount of the processor resources, the task's progress cannot be directly measured as the CPU time the task runs onto a CPU because the progress

also depends upon the activity of the other tasks simultaneously running on the same processor.

In this Thesis, we have shown that current time-based CPU schedulers are not fair for CMP processors due to shared hardware resources. For instance, it may occur that a higher-priority task has less chance to use some processor resources than a lower-priority task. For this reason, we introduce the concept of effective CPU capacity share in order to represent the actual CPU capacity used by a task in its execution.

We have also proposed two models that allow time-based CPU schedulers to take into account the CPU capacity enjoyed by each task when deciding future task schedules. Our results show that these models obtain similar performance to time-based CPU schedulers, while significantly improving fairness.

7.2 Future Work

The work done in this Thesis opens several research lines from which we highlight the following:

- **Energy accounting in multi-threaded processors**: The increasing number of shared hardware resources in MT processors introduces complexity in order to derive the per-task energy metering and accounting. It is an important challenge because the energy is an expensive resource in computing systems such as data centers and mobile devices.

- **CPU accounting in parallel task**: In the last years, parallel tasks have been more and more common in systems. In this Thesis, we have described how our CPU accounting mechanisms can be implemented in order to work in multi-threaded workloads, but they have not been evaluated in these workloads. One challenge is to evaluate our CPU accounting mechanisms in these workloads. An other interesting challenge can be how the operating system can combine the different CPU accounting of all threads of a same parallel task in order to obtain the CPU accounting of the parallel task.

- **Even-share accounting for SMT processors**: In this Thesis, a CPU accounting mechanism has been studied for CMP processors in even-share accounting. In SMT processors, the number of shared hardware resources is higher, which makes defining an even-share accounting and a CPU accounting mechanism more challenging in these processors.

- **Off-chip resources**: In this Thesis, we focus on on-chip hardware resources as a source of interference between tasks, which affects measuring the CPU capacity in MT processors. Also, off-chip resources such as network-on-chip or memory bus bandwidth can contribute to this interference. A good challenge is extending our CPU accounting mechanisms in order to take into account these resources as possible sources of interference.

- **The Principle of Accounting for future manycore systems**: Nowadays, heterogeneous processors are growing in the market. They consist of either several cores with different computation capacities or several cores and various accelerators such as graphics processing units, with different instructions set architectures. In these processors, it is interesting to study if the Principle of Accounting is guaranteed.

- **Load balancing**: The OS scheduling algorithm distributes the number of tasks in the systems in the different runqueues (or virtual CPUs) in order to have all runqueues with the same or similar number of tasks. In this Thesis, our schedulers balance the CPU capacity in each runqueue in order to achieve fairness. An important challenge is how the CPU capacity is distributed among virtual CPUs. In other words, the distribution of the tasks in a system depends upon the total CPU capacity of each runqueue.

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

7.3 Publications

In this section, we present a list of our research articles that are accepted for publications at conferences and journals. We also list the titles of the posters that are used to present our work at conferences and forums.

7.3.1 Conferences

- Carlos Luque, Miquel Moreto, Alexandra Fedorova, and Francisco J. Cazorla. *CPU Capacity-Aware Scheduling in Multicore Processors*. Submitted for publication to the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2014.

- Carlos Luque, Miquel Moreto, Francisco J. Cazorla, Roberto Gioiosa and Mateo Valero. *ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs*. In XXI Jornadas de Paralelismo. Valencia, Spain, September 2010.

- Carlos Luque, Miquel Moreto, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu and Mateo Valero. *ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs*. In International Conference on Parallel Architectures and Compilation Techniques (PACT). Raleigh, North Carolina. September 2009.

7.3.2 Journals

- Carlos Luque, Miquel Moreto, Francisco J. Cazorla and Mateo Valero. *Fair CPU Time Accounting in CMP+SMT Processors*. In Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers. Volume 9 Issue 4, January 2013, Article No. 5. This publication was selected to be presented at the HiPEAC conference held in Berlin, Germany, in January 2013.

- Carlos Luque, Miquel Moreto, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu and Mateo Valero. *CPU Accounting for Multicore Processors*. IEEE Transactions on Computers. Volume 61 Issue 2, February 2012.

- Carlos Luque, Miquel Moreto, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu and Mateo Valero. *CPU accounting in CMP Processors*. In IEEE Computer Architecture Letters. Volume 9 (1), April, 2009.

7.3.3 Posters

- Carlos Luque, Miquel Moreto, Francisco J. Cazorla and Mateo Valero. *CPU Accounting in the Multi-core and Multi-threaded Era*. In International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC), Berlin, Germany, January 2013

- Carlos Luque, Miquel Moreto, Francisco J. Cazorla and Mateo Valero. *CPU Accounting in the Multi-core and Multi-threaded Era*. In Proceedings of 3rd Barcelona Forum on Ph.D. Research in Information and Communication Technologies. Barcelona. Spain. October 2012.

7.3.4 Video

- Computer Now journal
  - http://www.computer.org/portal/web/computingnow/0212/whatsnew/tc

Bibliography

[1] C. Acosta et al. The MPsim Simulation Tool. Technical Report UPC-DAC-RR-CAP-2009-15, Computer Architecture Dept., UPC, 2009.

[2] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. Technical Report MIT/LCS/TM-450, MIT, 1991.

[3] Amazon. Amazon Elastic Compute Cloud (EC2) Website. http://aws.amazon.com/ec2/.

[4] R. Arndt, B. Sinharoy, S. Swaney, and K. Ward. Method and apparatus for frequency independent processor utilization recording register in a simultaneously multi-threaded processor, January 2011. US Patent 7,870,406.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[6] H. Beker and F. Piper. *Cipher systems: the protection of communications*. Wiley-Interscience, 1982.

[7] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[8] M. Broyles. IBM EnergyScale for POWER7 Processor-Based Systems. Technical report, IBM, 2011.

[9] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Architectural Support for Real-time Task Scheduling in SMT Processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES, 2005.

[10] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An Experimental Time-sharing System. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE, 1962.

[11] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: A System for Energy Efficient Computing in Virtualized Environments. In *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED, 2009.

[12] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2010.

[13] S. Eyerman and L. Eeckhout. Per-thread Cycle Accounting in SMT Processors. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2009.

[14] S. Eyerman and L. Eeckhout. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2010.

[15] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2006.

[16] A. Fedorova, M. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT, 2007.

[17] M. S. Floyd, S. Ghiasi, T. W. Keller, K. Rajamani, F. L. Rawson, J. C. Rubio, and M. S. Ware. System Power Management Support in the IBM POWER6 Microprocessor. *IBM J. Res. Dev.*, 51(6):733–746, November 2007.

[18] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. D. Maciel, and C. Thirumalai. *Advanced POWER Virtualization on*

IBM eServer p5 Servers: Architecture and Performance Considerations. IBM Redbook, 2005.

[19] R. H. Halstead, Jr. and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA, 1988.

[20] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 5th edition, 2011.

[21] IBM. Understanding CPU Utilization on AIX. https://www.ibm.com/developerworks/community/wikis/home, 2012.

[22] IBM. CPU frequency monitoring using lparstat. https://www.ibm.com/developerworks/community/wikis/home, 2013.

[23] Intel. Intel Atom Processor Z5xx Series. http://download.intel.com/design/processor/datashts/319535.pdf, June 2011. Datasheet.

[24] ITRS. International Technology Roadmap for Semiconductors. http://www.itrs.net, 2011.

[25] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, 2007.

[26] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: Cache Replacement and Utility-aware Scheduling. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2012.

[27] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA, 2004.

[28] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2004.

[29] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[30] H. Levy, J. Lo, J. Emer, R. Stamm, S. Eggers, and D. Tullsen. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA, 1996.

[31] T. Li, D. Baumberger, and S. Hahn. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2009.

[32] K. Luo, J. Gummaraju, and M. Franklin. Balancing Thoughput and Fairness in SMT Processors. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2001.

[33] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. ITCA: Inter-task Conflict-Aware CPU Accounting for CMPs. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2009.

[34] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. CPU Accounting for Multicore Processors. *IEEE Trans. Comput.*, 61(2):251–264, February 2012.

[35] C. Luque, M. Moreto, F. J. Cazorla, and M. Valero. Fair CPU Time Accounting in CMP+SMT Processors. *ACM Trans. Archit. Code Optim.*, 9(4):50:1–50:25, January 2013.

[36] P. Mackerras, T. S. Mathews, and R. C. Swanberg. Operating System Exploitation of the POWER5 System. *IBM J. Res. Dev.*, 49(4/5):533–539, July 2005.

[37] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[38] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: A QoS Framework for CMP Architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, April 2009.

[39] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A tool to understand large caches. Technical Report HPL-2009-85, HP, 2009.

[40] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2007.

[41] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual Private Caches. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA, 2007.

[42] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-chip Multiprocessor. *SIGPLAN Not.*, 31(9):2–11, September 1996.

[43] Oracle. White Paper. Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B Server Architecture, February 2012.

[44] J. K. Ousterhout. Scheduling techniques for concurrent systems. *ICDCS*, pages 22–30, 1982.

[45] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.

[46] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA, 2007.

[47] R. V. Polanczyk. Extending the Semantics of Scheduling Priorities. *Commun. ACM*, 55(8):48–52, August 2012.

[48] M. Qureshi. Adaptive Spill-Receive for robust high-performance caching in CMPs. In *IEEE 15th International Symposium on High Performance Computer Architecture*, HPCA, 2009.

[49] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[50] S. E. Raasch and S. K. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2003.

[51] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power Management Architecture of the 2nd Generation Intel Core microarchitecture, formerly codenamed Sandy Bridge. *Hot Chips*, August 2011.

[52] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.

[53] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2004.

[54] L. A. Seznec, A. Seznec, and T. Lafage. Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream. In *Workload Characterization of Emerging Applications*, 2000.

[55] A. Silberschats, P. Gailvin, and G. Gagne. Operating System Concepts, 7th edition.

[56] B. Sinharoy et al. IBM POWER7 multicore server processor. *IBM J. Res. Dev.*, 55:191–219, May 2011.

[57] B. Smith. Architecture And Applications Of The HEP Multiprocessor Computer System. *Fourth Symposium on Real Time Signal Processing*, 1981.

[58] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2000.

[59] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, 2002.

[60] Standard Performance Evaluation Corporation. SPEC CPU 2000 benchmark suite. http://www.spec.org.

[61] Standard Performance Evaluation Corporation. SPEC CPU 2006 benchmark suite. http://www.spec.org.

[62] S. Storino, A. Aipperspach, J. B. R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A commercial multithreaded RISC processor. In *45th International Solid-State Circuits Conference*, 1998.

[63] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA, 2002.

[64] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA, 1995.

[65] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT, 2013.

[66] D. W. Wall. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 1991.

[67] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA, 2003.

[68] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2010.

[69] S. Zhuravlev, S. Blagodurov, and A. Fedorova. AKULA: a toolset for experimenting and developing thread placement algorithms on multicore systems. In *PACT*, pages 249–260, 2010.

List of Figures

List of Tables

Acronyms

AR Accounting Register. 40

ATD Auxiliary Tag Directory. 38

BBV Basic Block Vector. 19

BCCS Balanced CPU Capacity scheduler. 97

BIC Bayesian Information Criterion. 19

BTB Branch Target Buffer. 70

CA Classical Approach. 33

CAA Cycle Accounting Architecture. 66

CFS Completely Fair Scheduler. 91

CGMT Coarse-Grain MultiThreading. 2

CMP Chip MultiProcessors. 2

CPU Central Processing Unit. 3

DVFS Dynamic Voltage and Frequency Scaling. 65

ECCS Equal CPU Capacity scheduler. 97

FGMT Fine-Grain MultiThreading. 2

HRSI Hardware Resource Status Indicator. 34

I^{2}TCA Improved InterTask Conflict-Aware. 8

ILP Instruction-Level Parallelism. 1

IPC Instruction Per Cycle. 9

isol Isolation. 69

ITCA InterTask Conflict-Aware. 8

LFSR Linear Feedback Shift Register. 73

LLC Last level Cache. 2

LRU Least Recently Used. 46

MIBTA Micro-Isolation Based Time Accounting. 8

MLP Memory Level Parallelism. 37

MSHR Miss Status Hold Register. 39

MT Multi-Threaded. 2

OS Operating System. 2

PHT Pattern History Table. 70

PTk Principal Task. 22

PURR Processor Utilization of Resources Register. 62

RFR Register File Release. 71

ROB Re-Order Buffer. 14

RSA Randomized Sampled Auxiliary Tag Directory. 9

RT Real Time. 94

sATD Sampled Auxiliary Tag Directory. 44

SMP Symmetric MultiProcessing. 1

SMT Simultaneous Multithreading. 1

SPEC Standard Performance Evaluation Corp.. 16

SPURR Scaled Processor Utilization of Resources Register. 66

ST Single-Threaded. 1

STk Secondary Task. 22

TLB Translation Lookaside Buffer. 16

TLP Thread-Level Parallelism. 1

TUS Task Under Study. 69

VM Virtual Machines. 87