CPU Accounting for Multicore Processors

Carlos Luque, Miquel Moreto, Francisco J. Cazorla, Roberto Gioiosa,
Alper Buyuktosunoglu, *Senior Member*, *IEEE*, and Mateo Valero, *Fellow*, *IEEE*

Abstract—In single-threaded processors and Symmetric Multiprocessors the execution time of a task depends on the other tasks it runs with (the workload), since the Operating System (OS) time shares the CPU(s) between tasks in the workload. However, the time accounted to a task is roughly the same regardless of the workload in which the task runs in, since the OS takes into account those periods in which the task is not scheduled onto a CPU. Chip Multiprocessors (CMPs) introduce complexities when accounting CPU utilization, since the CPU time to account to a task not only depends on the time that the task is scheduled onto a CPU, but also on the amount of hardware resources it receives during that period. And given that in a CMP hardware resources are dynamically shared between tasks, the CPU time accounted to a task in a CMP depends on the workload it executes in. This is undesirable because the same task with the same input data set may be accounted differently depending on the workload it executes. In this paper, we identify how an inaccurate measurement of the CPU utilization affects several key aspects of the system such as OS statistics or the charging mechanism in data centers. We propose a new hardware CPU accounting mechanism to improve the accuracy when measuring the CPU utilization in CMPs and compare it with the previous accounting mechanisms. Our results show that currently known mechanisms lead to a 16 percent average error when it comes to CPU utilization accounting. Our proposal reduces this error to less than 2.8 percent in a modeled 8-core processor system.

Index Terms—CPU accounting, chip-multiprocessor, shared last level of cache, cache partitioning algorithms.

1 Introduction

The Operating System (OS) provides the user with an abstraction of the hardware resources of the processor. The user application perceives this abstraction as if it were using the complete hardware while, in fact, the OS shares hardware resources among all the user applications. 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, single-threaded uniprocessor and Symmetric MultiProcessors (SMP) systems, the execution time of an application is influenced by the amount of hardware resources shared with the other running applications. It is also affected by how long the application runs with other applications. However, *the time accounted to that application is roughly the same regardless of the workload*[1] *in which it is executed, i.e., regardless of how many applications are sharing the hardware resources at any given time*. We call this principle, the *principle of accounting*. Unix-like systems differentiate the real execution time and the time an application actually is running on 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 application; *user* is the time spent by the application in the *user mode*; and *sys* is the time spent in the *kernel mode* on behalf of the application. In these systems, *sys+user* time is the execution time accounted to the application.

Fig. 1 shows the total (*real*) and the accounted execution time (*sys+user*) of the 171.*swim* (or simply *swim*) SPEC CPU 2,000 benchmark [21] 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, 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 Fig. 1 apply to other current Chip Multiprocessors (CMPs)). There are four cores in the chip, on which we run Linux 2.6.18. We move all 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 applications in the same core, its real execution time

- *C. Luque is with the Computer Architecture Department (DAC), Universitat Politècnica de Catalunya (UPC) and also with the Barcelona Supercomputing Center (BSC), C. Jordi Girona 29, Office 212, Nexus II, Barcelona 08034, Spain. E-mail: carlos.luque@bsc.es.*
- *M. Moreto and M. Valero are with the Computer Architecture Department (DAC), Universitat Politècnica de Catalunya (UPC), and the Barcelona Supercomputing Center (BSC), C. Jordi Girona 1-3, Campus Nord, Barcelona 08034, Spain. E-mail: {mmoreto, mateo}@ac.upc.edu.*
- *F.J. Cazorla is with the Spanish National Research Council (IIIA-CSIC) and the Barcelona Supercomputing Center (BSC), C. Jordi Girona 29, Office 212, Nexus II, Barcelona 08034, Spain. E-mail: francisco.cazorla@bsc.es.*
- *R. Gioiosa is with the Barcelona Supercomputing Center (BSC), C. Jordi Girona 29, Office 212, Nexus II, Barcelona 08034, Spain. E-mail: roberto.gioiosa@bsc.es.*
- *A. Buyuktosunoglu is with the Thomas J. Watson Research Center, 1101 Kitchawan Rd, Route 134/PO Box 218, Yorktown Heights, NY 10598. E-mail: alperb@us.ibm.com.*

Manuscript received 14 Oct. 2010; revised 20 July 2011; accepted 20 July 2011; published online 8 Aug. 2011.
Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-10-0565.
Digital Object Identifier no. 10.1109/TC.2011.152.

1. A workload is a set of applications running, simultaneously, in a system.

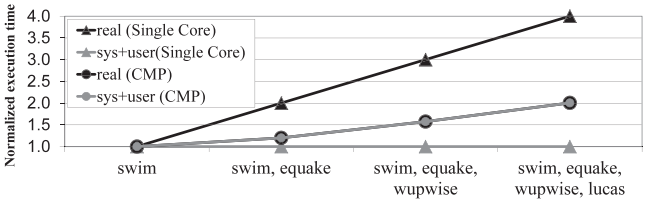


Fig. 1. Total (*real*) and accounted ($sys + user$) time of *swim* in different workloads running on an Intel Xeon Quad-Core CPU.

increases up to $4\times$ due to task switches forced by the OS (black triangles in Fig. 1). Nevertheless, *swim* is accounted roughly the same time (gray triangles), which is the time the application actually uses the CPU. Applications may suffer some delay because of cache data eviction and the lost of TLB contents caused by a process switch, but this effect is small in this case. Hence, even if *swim*'s total execution time increases depending on the other applications it is coscheduled with, the time accounted to *swim* is always the same.

In single-threaded uniprocessor systems, each running application uses 100 percent of the processor's resources and its progress can be generally measured in terms of the time spent running on the CPU. We call this approach the *Classical Approach* (CA). The CA has been proved to work well for single-threaded uniprocessor and SMP systems,[2] as the amount of hardware resources spatially shared is limited.

However, processors with shared on-chip resources, such as CMPs [9], make CPU accounting more complex because the progress of an application depends on the activity of the other applications running at the same time. Current accounting approaches may lead to inaccuracy for the time accounted to each application. 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 applications, as shown by the x-axis in Fig. 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 (cores) in the system. In Fig. 1, the gray circles show a variance up of to $2\times$ in the time *swim* is accounted for depending on the workload in which it runs. This means that (at least for Linux) *an application running on a CMP processor may be accounted different CPU utilization depending to the other applications running on the same chip at the same time*. From the user point of view this is an undesirable situation, as the same application with the same input data set is accounted differently depending on the applications it is coscheduled with.

The inaccuracy measuring per-task CPU utilization may affect several key components of a computing system, such as several commonly used programs (i.e., `top` or `time`) which may not properly account applications' progress. Finally, CPU accounting can be also used in data centers to charge users (together with other factor such as used amount of memory, disk space, I/O activity, etc.), according to their effective use of the system.

2. SMPs are systems with several single thread, single core chips. SMP systems still share other off-chip resources such as the memory bandwidth or the I/O channels. In this paper, we consider those shared resources less critical and only focus on on-chip shared resources.

The main contributions of this paper are:

1. We provide, for the first time, a comprehensive analysis of the CPU accounting accuracy of the CA. To the best of our knowledge, the CA is the only accounting mechanism for CMPs for open source OSs such as Linux.
2. Next, we propose a hardware mechanism, *Intertask Conflict-Aware* (ITCA) accounting [12], [13], which improves the accuracy of the CA for CMPs. When running on a modeled 2, 4, and 8-core CMP, ITCA reduces the inaccuracy of the CA from 7.0, 13, and 16 percent, to 2.4, 3.7, and 2.8 percent, respectively. Furthermore, we evaluate the accuracy of ITCA in conjunction with dynamic Cache Partitioning Algorithms (CPAs) [18].
3. ITCA [13] has been developed based on our understanding of the processor architecture. In this paper, 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 with small changes we can implement an improvement of ITCA, denoted $\mathrm{I^2TCA}$, that reduces the average off estimation of ITCA, mainly in the five worst workloads: from 32 to 13 percent in the 2-core configuration, from 35 to 17 percent in the 4-core configuration, and from 20 to 14 percent in the 8-core configuration.
4. We propose two different CMP accounting approaches: *full-share* accounting and *fair-share* accounting. In this paper, we adapt ITCA to consider both accounting approaches.

The rest of the paper is organized as follows: Section 2 analyzes and formalizes the CPU accounting problem. Section 3 describes our proposal of CPU accounting with improved accuracy in CMPs. The experimental methodology and the results of our simulations are presented in Section 4. Section 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 when changing the reference execution time to compute CPU utilization of a system in Section 6. Section 7 studies other issues regarding CPU accounting such as accounting for multithreaded applications. Section 8 discusses related work and, finally, Section 9 concludes the paper.

2 Formalizing the Problem

Currently, the OS perceives different cores in a CMP as multiple independent virtual CPUs. When it comes to CPU time accounting, the OS does not consider the interaction between tasks caused by shared hardware resources. With the CA, the OS only makes use of the time each task is scheduled onto a CPU. However, the time a task runs on a virtual CPU is not an accurate measure of the CPU time it has to be charged for. In fact the CPU time to account to a task in a CMP processor also depends on the amount of resources the task receives. In our view, CMP processors have to maintain the same *principle of accounting* that rules

today in uniprocessor and SMP systems: the CPU accounting of a task should be independent from the workload in which the task runs.

In this paper, we propose two accounting approaches that follow the principle of accounting: *full-share* accounting and *fair-share* accounting. In the former [13], when a task runs in a CMP, we consider that the time it should be accounted is the time it would take this task to run in the CMP in isolation. In the fair-share proposal, we consider that a task should be accounted for the time it takes the task to finish its execution when using an even part of the CMP shared hardware resources. In this paper, we mainly focus on the full-share accounting scheme and propose several techniques for its implementation. However, in Section 6 we show how to adapt our main proposal to implement the fair-share accounting scheme.

Let assume that a task X runs in a CMP for a period of time TR^{CMP}_{X,I_X}, in which it executes I_X instructions. In the full-share accounting approach, we consider that the actual time to account this task, denoted TA^{CMP}_{X,I_X}, should be the time it would take this task to execute these I_X instructions in isolation, denoted TR^{ISOL}_{X,I_X}. Equation 1 expresses the relative progress that task X has in this interval of time (P^{CMP}_{X,I_X}). The relative progress can also be expressed with the (2)

$$P^{CMP}_{X,I_X} = \frac{TR^{ISOL}_{X,I_X}}{TR^{CMP}_{X,I_X}}, \tag{1}$$

$$P^{CMP}_{X,I_X} = \frac{IPC^{CMP}_{X,I_X}}{IPC^{ISOL}_{X,I_X}}, \tag{2}$$

in which IPC^{CMP}_{X,I_X} and IPC^{ISOL}_{X,I_X} are the IPC of task X when executing the same I_X instructions in the CMP and in isolation, respectively. Then,

$$TA^{CMP}_{X,I_X} = TR^{CMP}_{X,I_X} \cdot P^{CMP}_{X,I_X}, \tag{3}$$

from which we conclude

$$TA^{CMP}_{X,I_X} = TR^{ISOL}_{X,I_X}. \tag{4}$$

This accounting decision follows our principle of workload-independent accounting.

When using this approach to measure CPU utilization, the main issue to address is how to determine dynamically (while a task X is simultaneously running with other tasks) on each task switch, the time (or IPC) it would have taken X to execute the same instructions if it had run alone in the system. An intuitive solution to this problem is to provide hardware mechanisms to determine the IPC in isolation of each task running in a workload by periodically running each task in isolation [3], [10]. By averaging the IPC in the different isolation phases, an accurate measurement of the full IPC (IPC in isolation) of the task can be obtained. However, as the number of tasks simultaneously executing in a CMP increases to dozens or even hundreds, this solution will not scale, as the number of isolation phases increases linearly with the number of tasks in the workload. As a consequence, the time the task runs in the CMP is reduced, affecting the system performance.

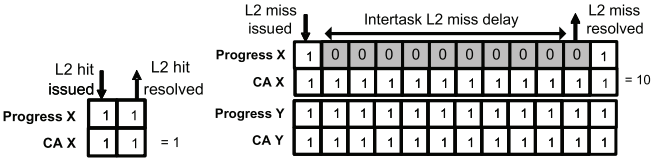


Fig. 2. Synthetic example to illustrate overestimation with the CA. The example highlights the effect of an intertask L2 miss on a 2-core CMP.

2.1 The Classical Approach

Throughout this paper, 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.

The CA accounts tasks based on the time they run on 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 CMP, which leads to intertask conflicts. 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 *overestimation*

$$TA^{CA}_{X,I_X} = TR^{CMP}_{X,I_X} > TR^{ISOL}_{X,I_X}. \tag{5}$$

A task has no overestimation only if it executes with no slowdown in CMP with respect to its execution in isolation, in which case

$$TA^{CA}_{X,I_X} = TR^{CMP}_{X,I_X} = TR^{ISOL}_{X,I_X}. \tag{6}$$

In order to illustrate the concept of overestimation and without loss of generality, we assume a dual-core in-order processor. The two cores share the L2 cache, while the first level data and instruction caches are private to each core. We further assume an L2 miss latency of 10 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 Section 4, all these assumptions are removed.

In Fig. 2, 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 1. Otherwise, it is marked as 0. The values in the CA row show the CPU time accounted to each task. Fig. 2a 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 accounted for 1 cycle for processing the memory access.

Fig. 2b 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

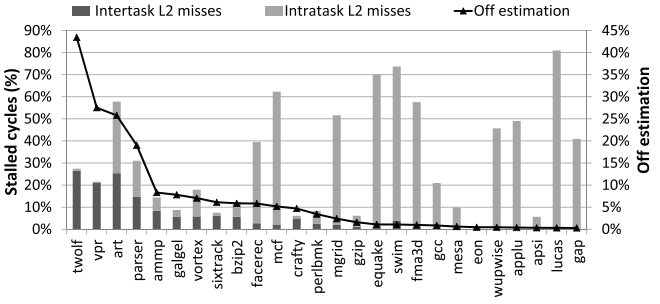


Fig. 3. Correlation between overestimation and stalled cycles due to intratask and intertask L2 misses in the CA.

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 10 cycles later. Under this scenario, X takes longer to serve the memory access and is accounted for 10 cycles. In this particular example, the intertask resource conflict causes an overestimation 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 overestimation in our CMP baseline architecture 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 overestimation in CPU utilization and intertask L2 misses in the CA. We run all possible 2-task workloads from SPEC CPU 2,000 benchmarks. For each workload, we compute the CPU utilization according to the CA, that is the execution time in CMP. For each task, we obtain the percentage of stalled cycles due to intratask and intertask L2 misses. Next, we sort the tasks in decreasing order according to the off estimation introduced by the CA, as shown in Fig. 3. We observe that the nine tasks with more than 5 percent stalled cycles due to intertask L2 misses are the nine 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 nine tasks with less than one percent stalled cycles due to intertask L2 misses present less than 1.1 percent off estimation. Overall, we observe a clear influence of intertask L2 misses on the accuracy of the CA. To address this problem our ITCA accounting mechanism focuses on intertask L2 misses.

3 Intertask Conflict-Aware Accounting

The target of our proposal, *Intertask Conflict-Aware* accounting, is to accurately estimate the CPU time 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 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.

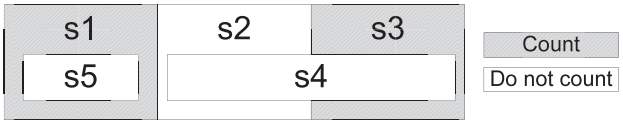


Fig. 4. Accounting decision for all possible states.

In the state (s1) we do a 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 ReOrder Buffer (ROB) and the register renaming stage is stalled, the task loses an opportunity to extract more Memory Level Parallelism (MLP). For example, assume that there are S instructions between the intertask L2 miss at 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 would 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 at the top of the ROB is an intertask L2 miss and the register renaming stage is stalled. We call this condition state (s4).

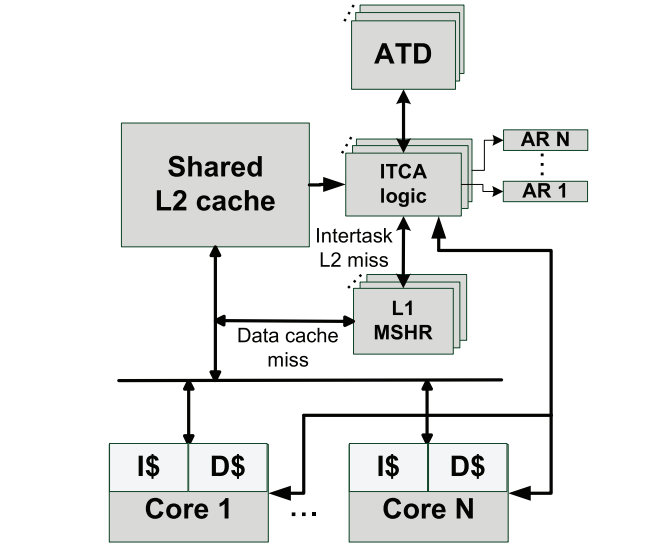
L2 instruction misses. ITCA also stops accounting to a task when the ROB is empty because of an intertask L2 cache instruction miss (s5). In our processor setup instruction cache misses do not overlap with other instruction cache misses. That is, at every instant, there is at most one in flight instruction 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.

Fig. 4 illustrates the accounting decision in the defined five states. 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).

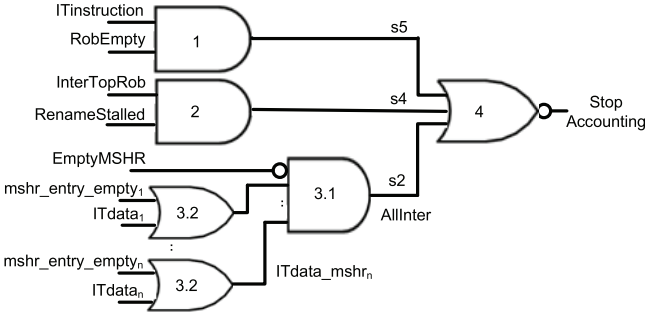
3.1 Hardware Implementation

Fig. 5 shows a sketch of the hardware implementation of our proposal, which makes use of several *hardware resource status indicators*. Next, we explain in depth the different parts of our approach.

Detecting intertask misses. We keep an *Auxiliary Tag Directory* (ATD) [18] for each core (see Fig. 5a). The ATD has the same associativity and size as the tag directory of the shared L2 cache and uses the same replacement policy. It stores the behavior of memory accesses per task in isolation.



(a) Baseline processor architecture



(b) Logic to stop accounting

Fig. 5. Hardware required for ITCA.

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 [8]. 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.

Accounting CPU time. We stop the accounting of a given task when:

1. The ROB is empty because of an intertask L2 cache instruction miss (gate (1) in Fig. 5b 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 cache data miss and we have a stall in the register renaming stage (gate (2) in Fig. 5b 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 misses (gates (3.1) and (3.2) in Fig. 5b 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 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.

In our processor setup, the memory bandwidth is not identified as a main source of interaction between tasks. We measured that in our setup, 90 percent of the workloads require less than 8 GB/s bandwidth, and that all of them require less than 12 GB/s. Hence, 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, this issue can be a problem. We leave this as part of our future work.

The cycles accounted to each task in each core are saved into a special purpose register per core, denoted *Accounting Register* or *AR* (see Fig. 5a), 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 [6]. On a task migration, both the ATD and the cache require some time to warm up but we expect this overhead to be low.

4 Experimental Results

4.1 Experimental Environment

Simulator. In order to compare ITCA and the CA, we use the MPsim simulator [1], a highly flexible cycle-accurate simulator that allows us to model CMP architectures. Our

TABLE 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
Mem latency/BW	300-cycle access, 12.1 GB/s

TABLE 2
Benchmarks' Cache Behavior (2 MB 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

baseline configuration is shown in Table 1, which represents an architecture with an 11-stage-deep pipeline. We use three processor setups: a 2-core CMP with a 2 MB L2 cache, a 4-core CMP with a 4 MB L2 cache, and an 8-core CMP with an 8 MB L2 cache.

Workloads. We feed our simulator with traces collected from the whole SPEC CPU 2,000 benchmark suite [21] using reference input sets. Each trace contains 300 million instructions, that are selected using SimPoint [20]. From these benchmarks, we generate 2-task, 4-task, and 8-task workloads. In each workload, the first task in the tuple is the *Principal Task* (PT) and the remaining tasks are considered *Secondary Tasks* (STs). In every workload, we execute the PT until completion. The secondary tasks are reexecuted until the PT completes. This allows us to characterize the accuracy of the CA and ITCA proposals based on the type of the PT and STs. It also allows us to easily compute the accuracy of each accounting mechanism by comparing the execution time of the PT when it runs in isolation with the predicted accounting time once the workload simulation ends.

We classify benchmarks into two groups depending on their cache behavior. 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 2. From these two groups, we generate different workload types denoted V_W, where V is the type of the PT and W is the type of the ST. We distinguish three combinations of ST: 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 four tasks, the PT is memory bound and the three STs are ILP bound. Note that for the 2-core configuration there is only one ST and, consequently, we can only evaluate STs 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.

Metrics. As the main metric, we measure how off is the estimation provided by each accounting mechanism. The *off estimation* (relative error of the approximation) compares the accounted time of a particular accounting approach for the PT with the actual time it should be accounted for. Equation (7) estimates the off estimation for a given accounting mechanism. Equation (8) provides the off estimation for the CA. For each accounting policy, we also report the average values of the five workloads with the worst off estimation, denoted *Avg5WOE*

$$\left|1 - \frac{TA^{CMP}_{PT,I_{PT}}}{TR^{ISOL}_{PT,I_{PT}}}\right|, \tag{7}$$

$$\left|1 - \frac{TR^{CMP}_{PT,I_{PT}}}{TR^{ISOL}_{PT,I_{PT}}}\right|. \tag{8}$$

4.2 Accuracy Results

Fig. 6 shows the off estimation of ITCA and the CA for our three processor setups. We show the average results of each group as we described in Section 4.1. The bars labeled AVG represent the average of each CMP configuration for all the groups. While on average the CA has an off estimation of 7.0 percent (2 cores), 13 percent (4 cores), and 16 percent (8 cores), ITCA reduces it to less then 2.4 percent (2 cores), 3.7 percent (4 cores), and 2.8 percent (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 percent (2 cores), 91 percent (4 cores), and 94 percent (8 cores) for the CA and only 32 percent (2 cores), 35 percent (4 cores), and 20 percent (8 cores) for ITCA.

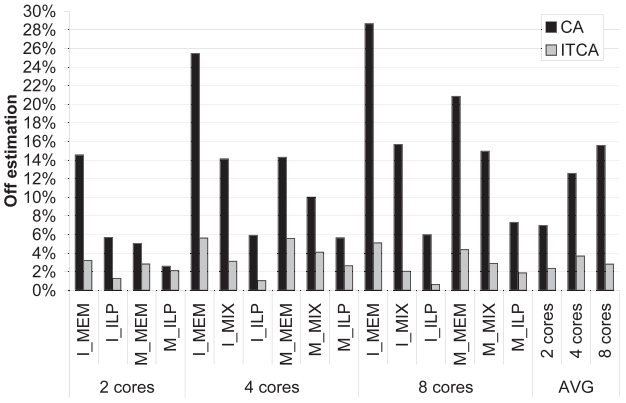


Fig. 6. Off estimation of the CA and ITCA for 2-, 4-, and 8-core CMPs with a shared 2, 4, and 8 MB L2 cache, respectively.

Next, we observe that the accuracy of the CA is worse when the PT is in the ILP group and all the STs 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 PT 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 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 (1 MB per core), the average off estimation of the CA increases from 7.0 (2 cores) to 16 percent (8 cores). The main reason for that behavior 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 percent, but reaches 29 percent in 8 cores.

Performance counters. Intuitively, one could think that with the performance counters that are present in current processors we can accurately approximate the CPU utilization of each task in a CMP architecture. As shown in [13], the best accounting mechanisms we can build with performance counters are much worse than ITCA. Our results report an average off estimation of 86 percent for the 8-core configuration. The main disadvantage of performance counters is that they do not measure the effect that one task can have on the other running tasks simultaneously.

4.3 Reducing the ATD's Overhead

The overhead of our baseline ATD is 30 KB per core (15-bit tag, 1,024 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 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 [18]. This scheme monitors only a subset of the cache sets (sampled sets), providing similar results to the ATD in terms of performance [18]. When using SATDs with ITCA, for accesses to nonsampled sets we cannot determine whether they are intertask or intratask misses. In this situation, ITCA does not consider these misses in the accounting task.

Fig. 7 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 two sets and the ATD has tags of 6 bits (denoted 6bitsTag-SD2). In this case, we reduce the size of the ATD to 6 KB, and increase the average off estimation and Avg5WOE to 5.2 and 40 percent, respectively. Recall that in this configuration, the CA leads to an average off estimation and Avg5WOE of 7.0 and 117 percent, respectively. Depending on the hardware budget available, different trade-offs are possible. For example, if 8 KB of area can be afforded, we can reduce the average off estimation and Avg5WOE to 2.9 and 40 percent, respectively.

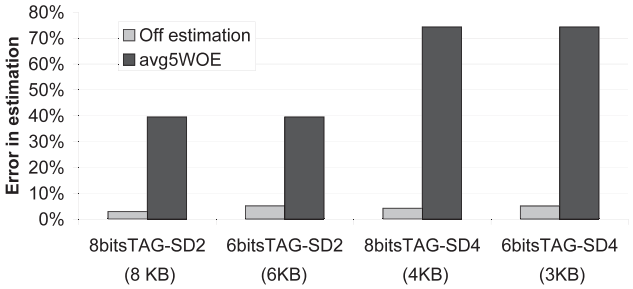


Fig. 7. Effect of reducing ATD overhead on accuracy.

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 paper. To that end, we use CACTI 6.5 [16] and assume a 32 nm technology. Our results show that the percentage that the ATD represents w.r.t the L2 cache is 2.9, 2.9, and 2.6 percent for the 2-, 4-, and 8-core configurations, respectively. The energy per access of the ATD is 4.0, 3.2, and 2.6 percent with respect to the L2 cache for the three configurations, respectively.

4.4 ITCA and Cache Partitioning Algorithms

Cache Partitioning Algorithms dynamically partition a shared cache among running tasks. CPAs significantly improve metrics such as throughput [10], [18], [22], fairness [11], [15], and Quality of Service [10], [15].

An accounting mechanism is also required in the presence of a CPA, 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 ITCA proposal can be applied to systems with CPAs 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 CPA when it suffers a miss in the L2 cache and a hit in its ATD. The ATD is already present in designs with CPAs and our accounting algorithm can make use of it. In such a design, the only hardware cost of ITCA is the logic shown in Fig. 5b.

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 CPAs

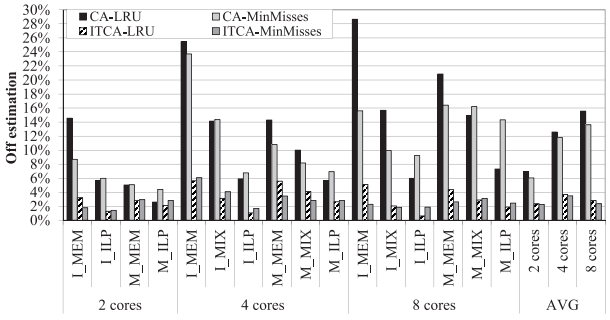


Fig. 8. Off estimation of the CA and ITCA with LRU and MinMisses dynamic CPA.

change the partition to adapt to the varying demands of competing tasks. We have chosen MinMisses algorithm [18], 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 Section 4.

Fig. 8 shows the off estimation of the CA and ITCA with LRU and MinMisses replacement policies. We observe that with LRU, the results are the same as the ones obtained in Section 4.2. The CA with LRU (denoted CA-LRU) obtains the worst results when the PT task has high ILP and any of the STs is memory bound. ITCA combined with LRU (denoted ITCA-LRU) performs better than CA-LRU, reducing the off estimation to 2.4 percent (2 cores), 3.7 percent (4 cores), and 2.8 percent (8 cores) on average, as we observed in Section 4.2.

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 PT, causing the PT 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 percent off estimation in the group I_MEM in four and eight 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 percent (2 cores), 12 percent (4 cores), and 14 percent (8 cores) to 2.2 percent (2 cores), 3.5 percent (4 cores), and 2.4 percent (8 cores). In comparison with CA-MinMisses, ITCA-MinMisses consistently reduces the off estimation to less than 6.0 percent in all groups and all configurations.

To sum up, the combination of ITCA with dynamic CPAs 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 Other Accounting Architectures

Eyerman and Eeckhout [4] propose a new cycle accounting architecture for SMT processors based on estimating the CPI stack [5] of each running task. In this section, we adapt this proposal to CMP processors and compare it to ITCA.

Eyerman's proposal tracks 15 different components of the CPI stack with dedicated hardware. The accounting architecture 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). This solution provides a 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 percent average prediction errors for 2- and 4-task workloads, respectively.

In [4], authors classify the execution cycles of each task when it runs in a SMT into three possible groups: *base*, *miss event*, or *waiting* cycles. Single-threaded execution time is then estimated as the sum of the base and miss event cycle components with some correction factors; the waiting cycle component represents the lost cycle count due to SMT execution. In single-threaded CMPs, this last component becomes zero and, consequently, the accounting time is estimated applying the correction factors for the shared resources to the corresponding cycle components.

Cycle accounting logic. The accounting architecture uses the notion of a dispatch slot (i.e., a 4-wide dispatch processor has four dispatch slots per cycle) and counts the number of per-task base and miss event dispatch slots. Dispatch slots per task can be classified into two possible situations in CMP architectures: 1) The task can dispatch an instruction. 2) The task cannot dispatch an instruction.

In the former situation, 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.

In the latter situation, if a task suffers a miss, we increase the corresponding miss counter. The cycle accounting keeps track of several possible miss sources: instruction L1 cache, instruction L2 cache, instruction TLB miss, full ROB (due to L2 data misses, data TLB misses, long latency units, dependencies, etc.). When there are several back-end misses, the accounting architecture gives priority to the miss associated to the first instruction in the ROB. When there are several front-end misses, we can only have a branch misprediction and an instruction L1, L2, or TLB miss. In this situation, the branch misprediction has more priority (only the branch misprediction counter is increased). If we have front-end and back-end misses, the front-end miss has more priority unless the ROB is full. In this situation, the back-end miss counter is increased.

In [4], nothing is said about other possibilities such as the task cannot dispatch but the ROB is not full and no miss is occurring. This could happen when the issue queues get full before the ROB or when there are no available rename registers. In this case, we decide to increase the other miss counter.

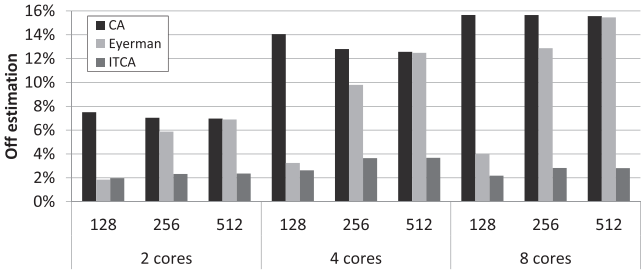


Fig. 9. ITCA and Eyerman's accounting architecture comparison across several processor configurations.

Finally, to obtain the execution time in isolation of the task, we add the base and miss counters. To take into account the effect of the CMP's shared resources, we correct the cycle component due to L2 data misses. To this end, we measure the MLP under CMP and estimate the MLP in isolation using a back-end miss event table (BMT). The ratio between these MLPs rescales the L2 cycle component. Finally, an ATD is used to obtain the increase in cache misses for the last level of cache. This multiplicative factor is also applied to the L2 cycle component. Since the branch predictor, the first level caches, and the TLBs are not shared, there is no correction factor for these CPI components.

Performance evaluation. Fig. 9 evaluates the accuracy of ITCA and Eyerman's proposal across multiple processor configurations. Eyerman's proposal assumes that the ROB is the main bottleneck of an out-of-order architecture. Their processor configuration models precisely this situation. To evaluate the sensitivity of their proposal, we have evaluated three configurations with different ROB sizes: 128, 256, and 512 entries.

With the smallest ROB size, Eyerman's proposal obtains its best results since the ROB is the main bottleneck for performance. In this configuration, an average off estimation of 1.9 percent (2 cores), 3.2 percent (4 cores), and 4.0 percent (8 cores) is obtained. However, when running on a configuration with larger ROB sizes, the average off estimation significantly increases, reaching 15 percent with a 512-entry ROB in eight cores. This off estimation is very close to the results obtained with the CA. The results shown in Fig. 9 indicate that not considering other important resources such as the issue queues or register files might lead to significant inaccuracies. In contrast, ITCA does not assume that the ROB is the main bottleneck for performance. This approach leads to more accurate results independently of the processor configuration in which it is run, ranging from 2.0 percent with a 128-entry ROB in two cores to 3.7 percent with a 512-entry ROB in four cores.

5 Improved ITCA (I^2TCA) Accounting Scheme

ITCA abstracts processor's architecture and takes into account only a few hardware resource status indicators as shown in Fig. 5b. 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 cache instruction miss. If both are active (gate 1), 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 cache data miss at the top of ROB. If both are active, we know that the core is stalled due to an intertask L2 cache data miss, so we stop accounting.
3. *AllInter* determines if all the active entries in the MSHR contain an intertask miss, that is whether there are only intertask L2 cache misses in flight (which corresponds to state (s2)), in which case ITCA stops accounting.

TABLE 3
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

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's approach 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*, 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 percent on average and always less than 0.5 percent). Therefore, those cycles do not significantly affect the final accuracy of ITCA and the corresponding HRSIs (*RobEmpty* and *ITinstruction*) are not considered in our brute force approach. Hence, we have only three HRSIs affecting the accuracy of ITCA: *RenameStalled*, *InterTopRob*, and *AllInter*.

Each row in Table 3 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 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 states [$\overline{RenameStalled}$,

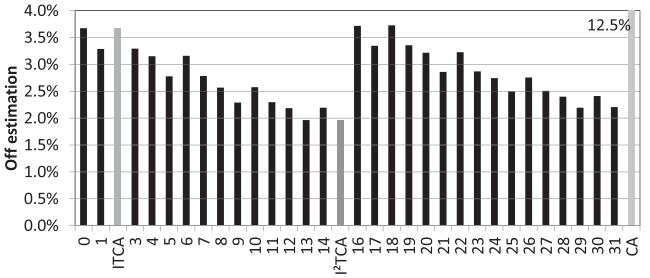


Fig. 10. Average off estimation of all the combinations for four cores and a 16-way 4 MB L2 cache.

$\overline{InterTopRob}$, $\overline{AllInter}$], [$\overline{RenameStalled}$, $InterTopRob$, $\overline{AllInter}$], and [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$].

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

Because of space constraints and without loss of generality, we do not present all the results of the design space exploration but focus on the most important conclusions. 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 5 HRSI states (6, 5, 3, 2, and 1 in Table 3), the accounting decision is not so clear. Fig. 10 shows the average off estimation for the remaining $2^5 = 32$ combinations in the 4-core configuration together with the CA. The same conclusions are derived for 2 and 8-core configurations. Accounting schemes are labeled according to the accounting decision taken in each of the 5 HRSI states. Thus, bar 0 shows the off estimation when in the remaining 5 HRSI states (6, 5, 3, 2, and 1) 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 percent, while the 32 accounting schemes are under 4.0 percent off estimation. ITCA corresponds to the third accounting scheme in Fig. 10. We observe that our original ITCA is quite close to the best observed combination, denoted *Improved ITCA*. The average off estimation for ITCA is 3.7 percent, while for I^{2}TCA it is 1.96 percent. The HRSI states in which ITCA and I^{2}TCA account cycles are shown in Table 3. We observe that both combinations account the cycles in HRSI states

$$[\overline{RenameStalled}, \overline{InterTopRob}, \overline{AllInter}],$$
$$[RenameStalled, \overline{InterTopRob}, \overline{AllInter}], \text{ and}$$
$$[\overline{RenameStalled}, InterTopRob, \overline{AllInter}].$$

In all these states there are always intratask L2 misses overlapping with intertask L2 misses in the MSHR, because

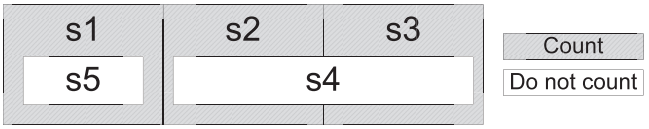


Fig. 11. I^{2}TCA accounting decision for all possible states.

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 state [$RenameStalled$, $InterTopRob$, $AllInter$] and [$RenameStalled$, $InterTopRob$, $\overline{AllInter}$] because those states are not possible when a task runs in isolation.

The main difference between ITCA and I^{2}TCA is the accounting decision for state (s2) explained in Section 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 Fig. 11, while ITCA does not, as shown in Fig. 4. 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 in Fig. 5b.

Next, we compare the average off estimation of the CA, ITCA, and I^{2}TCA in nine different processor configurations, as shown in Fig. 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 four cores with a 2 MB L2 cache is 21 percent, but it is 6.2 percent with an 8 MB 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 eight cores and a 2 MB L2 cache, I^{2}TCA reduces the off estimation down to 4.5 percent, while the CA and ITCA present a 31 and 8.6 percent 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 simultaneously increases. This behavior is similar in I^{2}TCA, but the off

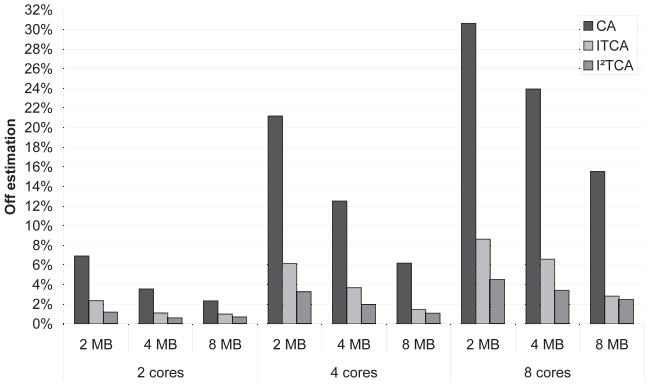


Fig. 12. Average off estimation for two, four, and eight cores and a 16-way 2, 4, and 8 MB L2 cache.

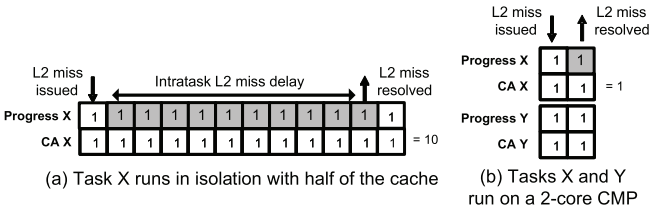


Fig. 13. Synthetic example for explaining underestimation with the CA and I^{2}TCA.

estimation is always under 1.2 percent (2 cores), 3.3 percent (4 cores), and 4.5 percent (8 cores) on average.

I^{2}TCA reduces the average off estimation of ITCA, mainly in the five worst workloads (not shown in Fig. 12): from 32 to 13 percent in the 2-core configuration, from 35 to 17 percent in the 4-core configuration, and from 20 to 14 percent 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.

6 Fair-Share Accounting

In the previous sections, we have assumed that 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. This is the natural extension of the concept of CPU utilization in single-threaded uniprocessor and SMP systems. Moreover, this accounting mechanism accomplishes with the *principle of accounting*, meaning that a task should be always accounted the same regardless of the workload in which it runs.

In this section, we explore a different accounting approach, *fair-share accounting*, which also accomplishes with the *principle of accounting*. We consider that a task should be accounted the time it would require to run with its fair share of resources, i.e., $1/N$th of the shared processor resources, where N is the number of cores. In the case of a CMP with a shared cache, we have to assign $1/N$th of the cache space to the task. This baseline accounting can be used together with recent cache management techniques developed for private last level caches [17]. In this section, we show that few changes to I^{2}TCA are enough to adapt it to the fair-share accounting without losing accuracy.

In the original reference accounting, ITCA corrects the case in which the CA approach leads to overestimation. 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 fair-share reference accounting, the CA not only suffers from overestimation, but it can also suffer from *underestimation*. For example, assume that when a task X runs with $1/N$th of the processor resources, a given load misses in the L2 cache. 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 fair-share accounting processor setup, leading to underestimation with the CA. This situation with a 2-core CMP is illustrated in Fig. 13, where 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 fair-share configuration (with $1/N$th 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 $1/N$th of the L2 cache. In this case, the task can progress faster than when it runs in the fair-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 $1/N$th of the shared resource. This is not taken into account with the original I^{2}TCA.

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

6.1 Hardware Requirements

We add two new hardware resource status indicators 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 $1/N$th to each core (remind that in the original 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. The second HRSI, denoted *IntraTopRob*, indicates if the oldest instruction in the ROB is an intratask L2 cache data miss. This signal is not useful with previous schemes, since in that state in the original 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 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 5. The signal *IntraTopRob* cannot be active when signals *AllInter* or *InterTopRob* 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 [$\overline{RenameStalled}$, $\overline{InterTopRob}$, $\overline{AllInter}$].

TABLE 4
States of a Task and Accounting Decision

RenameStalled	InterTopRob	AllInter	**InterHit**	**IntraTopRob**	**I²TCA**
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**

When the signal *InterHit* is active, a task is running faster than when it receives 1/Nth of the cache, and hence, I²TCA accounts two cycles in all states except [$RenameStalled$, $\overline{InterTopRob}$, $AllInter$], [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$], and [$RenameStalled$, $\overline{InterTopRob}$, $\overline{AllInter}$]. In the former two states, a task is progressing slower than when receiving its fair-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 fair-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 fair-share case and, consequently, I²TCA 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 fair-share case and I²TCA accounts one cycle to the task.

6.2 Accuracy Results

Fig. 14 shows the off estimation of I²TCA and the CA for our three processor setups. We show the average results of each group as we described in Section 4.1. The bars labeled 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 percent (2 cores) to 9.7 percent (8 cores). We also observe that the accuracy of I²TCA is better than the accuracy of the CA in all the groups. This is due to the fact that a task suffers both intertask L2 misses and intertask L2 hits during its execution. I²TCA takes into account this situation and hence, on average I²TCA reduces the off estimation down to 1.3 percent (2 cores), 2.5 percent (4 cores), and 3.9 percent (8 cores).

7 Other Considerations

In this paper, we have evaluated ITCA with multiprogrammed workloads (workloads composed of single-threaded tasks). ITCA also works, with minimal changes, for multithreaded workloads. The interaction between threads in a parallel application can be *positive* when, for example, one thread prefetches data for another thread. This behavior is intrinsic to the task, and hence it also occurs when running in isolation. When one multithreaded 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, an application identifier field has to be added to entries in the MSHR to detect intertask misses. This identifier has to be assigned to applications by the OS, and provides to the architecture when scheduling is decided. Common OSs, such as Linux, already use an identifier for the application (PID) and for the threads within the same application (TID). No other changes are required in the ITCA implementation.

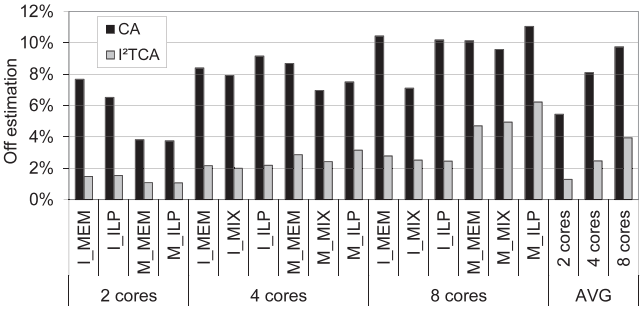


Fig. 14. Off estimation of the CA and I²TCA with the fair-share accounting for two, four, and eight cores and 16-way 2, 4, and 8 MB L2 caches.

With ITCA, the CMP processor reports a different accounting for each thread of a multithreaded 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 paper.

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

8 Related Work

We are not aware of any other work which studies CPU accounting for CMP architectures. Thus, ITCA is the first accounting mechanism for CMP processors. For SMT processors [19], [23], other proposals have been made. The IBM POWER5 processor (a dual-core and 2-context SMT processor) includes a per-task accounting mechanism called *Processor Utilization of Resources Register* (PURR) [13]. The PURR approach estimates the time of a task based on the number of cycles the task can decode instructions: each POWER5 core 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, both tasks running on the same core are accounted one half of cycle. An improvement of PURR, denoted *scaled PURR* (SPURR) [7], is implemented in the IBM POWER6 chip, which uses pipeline throttling and DVFS. SPURR provides a scaled count that compensates the impact of throttling and DVFS. ITCA can work in environments in which cores work at different frequencies with no change in its philosophy. The only effect seen by ITCA is a difference in the memory latency. Throttled cycles are simply not accounted to any task.

A recent patent [2] introduces a similar mechanism to PURR. The main difference with PURR is that the target SMT processor in [2] is able to decode instructions from up to two tasks per cycle, while the POWER5 can only decode instructions from one task per cycle. The main difference between PURR and this mechanism [2] is that, in the former, tasks are charged based on the number of decode cycles they use, regardless of the number of instruction they decode in each cycle. In the latter approach, the focus is put on the number of instructions a task dispatches. For example, if in a given cycle the first task dispatches five instructions and the second two, the first task is charged 5/7 and the second 2/7.

In the OS system domain, the most similar work to our proposal is presented by Fedorova et al. [6]. The authors 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. The proposed solution is divided into two components. During a sample phase the OS runs a task with all the possible corunners and uses a model to estimate the task's *Fair IPC*. This model extrapolates the Fair IPC of the task from the number of cache misses a task suffers when running with another task. During the scheduling phase, the OS scheduler increases or reduces the time quantum of the task in order to provide good performance isolation. The main difference with our work is that we do not use a model to estimate the isolated performance of a task but we provide hardware support to the OS in order to accurately account each task for the progress it makes. Once the accounting is available for a task, the OS scheduler proposed by Fedorova et al. [6] can be used on top of our mechanism to compensate the time quantum of tasks to meet their expected performance.

9 Conclusions

CMP architectures introduce complexities when measuring tasks' CPU utilization because the progress done by a task varies depending on the activity of the other tasks running at the same time. The current accounting mechanism, the CA, introduces inaccuracies when applied in CMP processors. In this paper, we present hardware support for a new accounting mechanism called *Intertask Conflict-Aware* 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 percent (8 cores) while the CA presents a 7.0, 13, and 16 percent 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 MinMisses scheme) with nearly no extra hardware addition, motivating the use of ITCA and CPAs simultaneously.

We 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 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 percent. Finally, we showed the effects of a change in the accounting approach (fair-share accounting) on the accuracy of different accounting mechanisms.

As a part of our future work we plan to explore CMP architectures in which each core is SMT. In this type of architectures we need to combine some of the solutions for SMT architectures with our ITCA proposal for CMP processors.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625 and grants BES-2008-003683 and JCI-2008-3688, by the HiPEAC Network of Excellence (IST-004408) and a Collaboration Agreement between IBM and BSC with funds from IBM Research and IBM Deep Computing organizations. The authors are grateful to Pradip Bose and Chen-Yong Cher from IBM for their technical support, to Enrique Fernández from the University of Las Palmas de Gran Canaria for his help setting up the Intel Xeon Quad-Core processor, to Jaume Abella for his comments about power consumption, and to the reviewers for their valuable comments.

References

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

[2] R.L. Arndt et al., "Method and Apparatus for Frequency Independent Processor Utilization Recording Register in a Simultaneously Multi-Threaded Processor," US Patent 7,870,406, to IBM Corp., Patent and Trademark Office, 2011.

[3] F.J. Cazorla et al., "Predictable Performance in SMT Processors: Synergy between the OS and SMTs," *IEEE Trans. Computers,* vol. 55, no. 7, pp. 785-799, July 2006.

[4] S. Eyerman and L. Eeckhout, "Per-Thread Cycle Accounting in SMT Processors," *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 133-144, 2009.

[5] S. Eyerman et al., "A Performance Counter Architecture for Computing Accurate CPI Components," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 175-184, 2006.

[6] A. Fedorova, M. Seltzer, and M. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," *Proc. 16th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT),* pp. 25-38, 2007.

[7] M.S. Floyd et al., "System Power Management Support in the IBM POWER6 Microprocessor," *IBM J. Research and Development,* vol. 51, no. 6, pp. 733-746, 2007.

[8] R.L. Mattson et al. "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, pp. 78-117, 1970.

[9] L. Hammond, B.A. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *Computer*, vol. 30, no. 9, pp. 79-85, Sept. 1997.
[10] R.R. Iyer et al., "QoS Policies and Architecture for Cache/Memory in CMP Platforms," *Proc. Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 25-36, 2007.
[11] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," *Proc. 13th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT)*, pp. 111-122, 2004.
[12] C. Luque et al., "CPU Accounting in CMP Processors," *Computer Architecture Letters*, vol. 8, no. 1, pp. 17-20, 2009.
[13] C. Luque et al., "ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs," *Proc. 18th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT)*, pp. 203-213, 2009.
[14] P. Mackerras, T.S. Mathews, and R.C. Swanberg, "Operating System Exploitation of the POWER5 System," *IBM J. Research and Development*, vol. 49, nos. 4/5, pp. 533-539, 2005.
[15] M. Moreto et al., "FlexDCP: A QoS Framework for CMP Architectures," *SIGOPS Operating Systems Rev.*, vol. 43, no. 2, pp. 86-96, 2009.
[16] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A Tool to Understand Large Caches," Technical Report HPL-2009-85, HP, 2009.
[17] M.K. Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," *Proc. 15th Int'l Conf. High-Performance Computer Architecture (HPCA)*, pp. 45-54, 2009.
[18] M.K. Qureshi and Y.N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," *Proc. 39th Int'l Symp. Microarchitecture (MICRO)*, pp. 423-432, 2006.
[19] M.J. Serrano, R. Wood, and M. Nemirovsky, "A Study of Multistreamed Superscalar Processors," Technical Report #93-05, UCSB, 1993.
[20] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *Proc. 10th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT)*, pp. 3-14, 2001.
[21] Standard Performance Evaluation Corporation, "SPEC CPU 2000 Benchmark Suite," http://www.spec.org, 2011.
[22] G.E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," *Proc. Eight Int'l Conf. High-Performance Computer Architecture (HPCA)*, pp. 117-128, 2002.
[23] D. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA)*, pp. 392-403, 1995.

Carlos Luque received the BS degree in computer science from the University of Las Palmas de Gran Canaria and MS degree in computer science from the Universitat Politècnica de Catalunya. He is a doctoral candidate in the Computer Architecture Department at the Universitat Politècnica de Catalunya (UPC) and a resident student at the Barcelona Supercomputing Center, Spain. His research interests include CPU accounting, multithreaded architectures, and operating systems.

Miquel Moreto received the BS and MS degrees in mathematics and electrical engineering from the Universitat Politècnica de Catalunya (UPC), Spain, and the PhD degree in 2010 in the Computer Architecture Department at the same university. Currently, he is a postdoctoral fellow at the International Computer Science Institute (ICSI), Berkeley. His research interests include studying shared resources in multithreaded architectures and modeling interconnection networks in parallel systems.

Francisco J. Cazorla is a researcher in the Spanish National Research Council. He leads the group on Computer Architecture/Operating System interface (CAOS) at the Barcelona Supercomputing Center in which he has participated and led several industrial and public-funded projects. His research area focuses on multithreaded architectures for both high-performance and real-time systems on which he is coadvising 10 PhD theses. He has coauthored more than 50 papers in international refereed conferences and journals. He is member of HIPEAC and the ARTIST Networks of Excellence.

Roberto Gioiosa received the MS and PhD degrees in computer science from the University of Rome, "Tor Vergata." Currently, he is a senior researcher in the group on Computer Architecture/Operating System interface (CAOS) at the Barcelona Supercomputing Center. His research interests include high-performance computing, operating systems, data centers, low-power, and reliable systems, real-time embedded systems. He has published more than 20 papers in international refereed conferences and journals.

Alper Buyuktosunoglu received the PhD degree in electrical and computer engineering from the University of Rochester. Currently, he is a research staff member in Reliability and Power-Aware Microarchitecture Department at IBM TJ Watson Research Center. He has been involved in research and development work in support of IBM p-series and z-series microprocessors in the area of power-aware computer architectures. His research interests are in the area of high-performance, power/reliability-aware computer architectures. He has over 35 pending/issued patents, has received several IBM-internal awards, has published more than 40 papers, and has served on various conference technical program committees in this area. He is a senior member of the IEEE and is currently serving on the editorial board of *IEEE MICRO*.

Mateo Valero has been a full professor in the Computer Architecture Department, Universitat Politècnica de Catalunya (UPC), since 1983. Since May 2004, he is the director of the Barcelona Supercomputing Center (the National Center of Supercomputing in Spain). His research topics are centered in the area of high-performance computer architectures. He has published approximately 500 papers, has served in the organization of more than 200 international conferences. His research has been recognized with several awards. Among them, the Eckert-Mauchly Award, Harry Goode Award, the "King Jaime I" in research and two National Awards on Informatics and on Engineering. He has been named Honorary Doctorate by the Universities of Chalmers, Belgrade, Las Palmas de Gran Canaria and Zaragoza in Spain and the University of Veracruz in Mexico. He is an academic of the Royal Spanish Academy of Engineering, a correspondent academic of the Royal Spanish Academy of Sciences, an academic of the Royal Academy of Science and Arts, and member of the Academia Europea. He is a fellow of the IEEE and the ACM and Intel Distinguished Research Fellow.