2009 18th International Conference on Parallel Architectures and Compilation Techniques

ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs

Carlos Luque*, Miquel Moreto†, Francisco J. Cazorla*, Roberto Gioiosa‡, Alper Buyuktosunoglu‡ and Mateo Valero*,†

*Barcelona Supercomputing Center (BSC), Barcelona, Spain. Email: {carlos.luque, francisco.cazorla}@bsc.es
†Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. Email: {mmoreto, mateo}@ac.upc.edu
‡IBM T.J. Watson Research Center, New York, USA. Email: {rgioios, alperb}@us.ibm.com

***Abstract*—Chip-MultiProcessor (CMP) architectures are becoming more and more popular as an alternative to the traditional processors that only extract instruction-level parallelism from an application. CMPs introduce complexities when accounting CPU utilization. This is due to the fact that the progress done by an application during an interval of time highly depends on the activity of the other applications it is co-scheduled with.**

In this paper, we identify how an inaccurate measurement of the CPU utilization affects several key aspects of the system like the application scheduling 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 19% average error when it comes to CPU utilization accounting. Our proposal reduces this error to less than 1% in a modeled 4-core processor system.

***Keywords*-Cycle Accounting; Chip-MultiProcessor; Cache Partitioning Algorithms; Fairness; ATD**

I. Introduction

The Operating System (OS) provides the user with an abstraction of the hardware resources. The user application perceives this abstraction as if it is using the complete hardware while, in fact, the OS shares hardware resources among 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, like the cache memory or the I/O bandwidth.

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 always 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 like 'time' or 'top' provide three values: *real*, *user* and *sys*. *Real* is the elapsed wall clock time between 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.

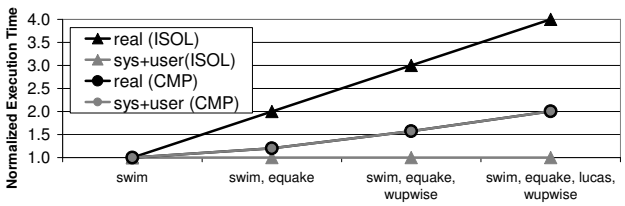


Figure 1: Total (*real*) and accounted (*sys+user*) time of *swim* in different workloads running on an Intel Xeon Quad-Core CPU

Figure 1 shows the total (*real*) and the accounted execution time (*sys+user*) of the 171.*swim* (or simply *swim*) SPEC CPU 2000 benchmark [1] when running in different workloads. In this figure, the time results are normalized to the real execution time of *swim* when it runs in isolation (ISOL). 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 all current CMPs), which has 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 increases up to 4x due to context switches done by the OS (black triangles). Nevertheless, *swim* is accounted roughly the same time (grey triangles), which is the time the application actually uses the CPU. Applications may suffer some delay because they lose part of the cache and TLB contents on every context 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 co-scheduled with, the time accounted to *swim* is always the same.

[1]A workload is a set of applications running, simultaneously, on a CPU

1089-795X 2009
DOI 10.1109/PACT.2009.33

In uniprocessor systems, each running application uses 100% of the processor's resources and its progress can be measured in terms of the time spent on the CPU. We call this approach the *Classical Approach* (CA). The CA has been proved to work well for uniprocessor and Symmetric MultiProcessors (SMP) systems[2], as the amount of hardware shared resources is limited. In these systems, the major labor of the OS scheduler is to time share the CPUs between applications.

However, processors with shared resources, like CMPs [2], 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 OSs still use the CA for multicore processors, which can 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. Next, we execute *swim* with several workloads as shown by the x-axis in Figure 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 less than the number of virtual CPUs (cores) in the system. In Figure 1, the grey circles show a variance up of to 2x in the time *swim* is accounted for depending on the workload in which it runs. This means that (at least with current known open source OSs like Linux) *an application running on a CMP processor may be accounted differently according 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 set is accounted differently depending on the applications it is co-scheduled with.

CPU accounting affects several key components of a computing system: First, if the OS scheduler does not properly account the CPU utilization of each application, the OS scheduling algorithm will fail to maintain fairness between applications. As a consequence, the scheduling algorithm cannot guarantee that an application progresses with its work as expected. Second, in data centers customers are charged according to the utilization of the CPU they use. Third, virtual machines are becoming very common and allow users to consolidate. Each virtual machine should be accounted for the correct number of resources it uses.

The main contributions of this paper are: For the first time, we provide a comprehensive analysis of the CPU accounting accuracy of the CA. To best of our knowledge CA is the only accounting mechanism for CMPs for open source OSs like Linux. Next, we propose a hardware mechanism, *Inter-Task Conflict-Aware* (ITCA) accounting [3], that improves the accuracy of the CA for CMPs. For all 676 pairs of SPEC CPU 2000 benchmark running in a 2-core CMP architecture, ITCA reduces the inaccuracy (off estimation) to 1% (20% in the worst five cases), while the CA presents an inaccuracy of 9% (120% in the worst five cases). For 64 4-task workloads running on a 4-core CMP processor, ITCA leads to an average inaccuracy of 1% (5% in the worst five cases) while the CA shows an inaccuracy of 19% (149% in the worst five cases). Furthermore, we evaluate the accuracy of ITCA in conjunction with both Static [4] and Dynamic [5] Cache Partitioning Algorithms (CPA). The combination of ITCA with dynamic CPAs significantly reduces the inaccuracy of the CA in conjunction with dynamic CPAs from 9% to 1%. Moreover, ITCA leverages the Auxiliary Tag Directories (ATDs) that are already used by current cache partitioning algorithms, like MinMisses [5], with nearly no extra hardware addition motivating the use of both schemes simultaneously.

The rest of the paper is organized as follows. Section II analyses and formalizes the CPU accounting problem. Section III describes our proposal of CPU accounting for improving the accuracy. The experimental methodology and results of our simulation are presented in Section IV. Section V evaluates the combination of our proposal with cache partitioning algorithms. Section VI studies other issues regarding CPU accounting such as fairness, performance-counter based accounting. Section VII discusses related work and finally Section VIII concludes the paper

II. Formalizing the Problem

Currently, the OS perceives different cores in a CMP as multiple independent virtual CPUs. The OS does not consider the interaction between tasks caused by shared resources in the CA. However, the time running on a virtual CPU is not an accurate measure of the amount of CPU resources the task has received. The CPU time to account to a task in a CMP processor does not only depend on the time that task is scheduled onto a CPU, but also on the progress it makes during that time. In our view, CMP processors, have to maintain the same *principle of accounting* that rules today in SMP and uniprocessor systems accounting: the CPU accounting of a task should be independent from the rest of the workload in which this task runs. For example, let's assume that a task X runs for a period of time in a CMP (TR^{CMP}_{X,I_X}), in which it executes I_X instructions. It is our position that the actual time to account this task, denoted as 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}. The relative progress that task X has in this interval of time (TR^{CMP}_{X,I_X}) can be expressed as

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

[2]SMPs are systems with several single thread, single core chips. SMP systems still share other off-chip resources like 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 relative progress can also be expressed as

$$P_{X,I_X}^{CMP} = \frac{IPC_{X,I_X}^{CMP}}{IPC_{X,I_X}^{ISOL}} \quad (2)$$

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

$$TA_{X,I_X}^{CMP} = TR_{X,I_X}^{CMP} \cdot P_{X,I_X}^{CMP} \quad (3)$$

from which we conclude

$$TA_{X,I_X}^{CMP} = TR_{X,I_X}^{ISOL} \quad (4)$$

This follows our principle of workload-independent accounting.

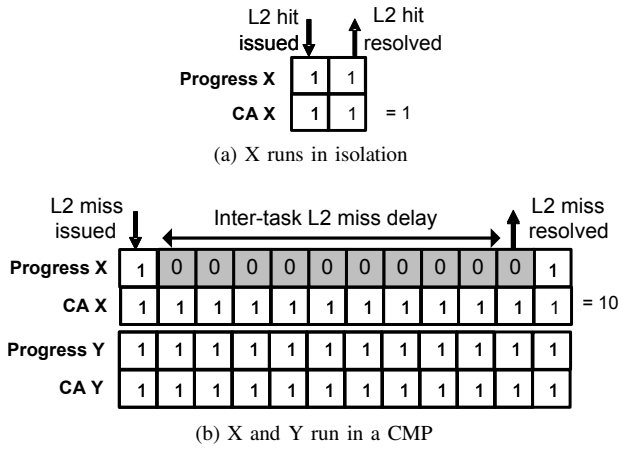
The main issue to address is how to determine dynamically (while a task X is simultaneously running with other tasks) on each context switch, the time (or IPC) it will take X to execute the same instructions if it is alone in the system. An intuitive solution to this problem is to provide hardware mechanisms to determine the IPC in isolation of each task running in a workload by periodically running each task in isolation [6][7]. By averaging the IPC in the different isolation phases, an accurate measurement of the IPC of the task can be obtained when running in isolation. However, as the number of tasks simultaneously executing in a multicore processor increase 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 CMP architectures is reduced, affecting the system performance.

A. The Classical Approach

Throughout this paper, we refer to *inter-task* 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 inter-task L2 cache miss when it accesses a line that was evicted by another task, but would have been in cache, if X had run in isolation. Likewise, *intra-task* 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 does. Therefore, the CA implicitly assumes that tasks have full access to the processor resources when running. However, each task shares resources with other tasks when running in a CMP which leads to inter-task 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 *over-estimation*

$$TA_{X,I_X}^{CA} = TR_{X,I_X}^{CMP} > TR_{X,I_X}^{ISOL} \quad (5)$$



(a) X runs in isolation

(b) X and Y run in a CMP

Figure 2: Synthetic example for explaining over-estimation with CA. The example highlights the effect of an inter-task L2 miss

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

$$TA_{X,I_X}^{CA} = TR_{X,I_X}^{CMP} = TR_{X,I_X}^{ISOL} \quad (6)$$

The main source of over-estimation in our CMP baseline architecture is inter-task conflicts and, in particular, inter-task L2 misses. In order to illustrate the concepts of over-estimation, we assume, for sake of simplicity and without loss of generality, 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 1 cycle. Even if this latency is not representative of any current processor, it is perfectly valid for the purpose of illustrating the problem of CPU accounting. For the purpose of illustration as well, we assume that the execution time of a task X when running in isolation, TR_{X,I_X}^{ISOL}, is known. In Section IV, all these assumptions are removed.

In Figure 2, each square represents a task cycle. The *Progress* row shows whether a task progresses. 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. Figure 2 (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 accounted for 1 cycle for processing the memory access.

Figure 2 (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 of X from the L2 cache, causing the previous L2 hit of X to become an inter-task L2 miss. This inter-task miss causes X to stall its execution (dark square) until it is resolved which is 10 cycles later. Under this scenario, X takes longer to serve

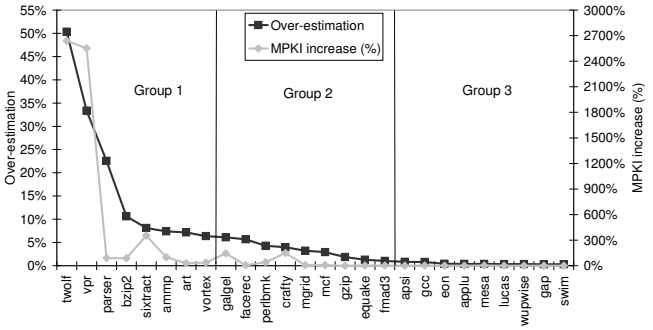


Figure 3: Correlation between over-estimation and inter-task L2 misses (MPKI) in the Classical Approach

the memory access and is accounted for 10 cycles. In this particular example, the inter-task 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 inter-task miss, doing the same progress as in isolation.

We derive the relationship between over-estimation and inter-task L2 misses in the CA from the following experiment: we run all 676 2-task workloads from SPEC CPU 2000 benchmarks. In each workload, we compute the accounting provided by the CA, that is, the execution time in CMP. For each task, we obtain the increment in the number of L2 misses a task suffers when it runs with another task instead of running in isolation. We compute this increment as

$$\Delta MPKI = \left| 1 - \frac{MPKI_{CMP}}{MPKI_{ISOL}} \right| \cdot 100 \qquad (7)$$

where $MPKI_{CMP}$ stands for L2 misses per thousand (kilo) instructions when the task runs together with another task in CMP mode and $MPKI_{ISOL}$ stands for L2 misses per thousand (kilo) instructions when it runs in isolation.

Next, we correlate the over-estimation provided by the CA with the increase in the L2 MPKI of each task. We sorted the tasks in decreasing order by the over-estimation introduced by the CA as shown in Figure 3. We observe that between the 8 tasks with the highest over-estimation (group 1) there are 7 of the tasks with the highest increase in MPKI. Analogously, the 9 tasks with the lowest over-estimation (group 3) are the 9 tasks with the lowest increase in MPKI. Finally in group 2, in which there are the 9 tasks with medium over-estimation, we find 7 of the tasks with average increase in the MPKI. This shows the influence of inter-task L2 misses on the accuracy of the CA.

III. Inter-Task Conflict-Aware Accounting

The target of our proposal is to accurately estimate the 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 inter-task L2 cache miss. In other words, a task is accounted CPU cycles when it is progressing or when it is stalled due to an intra-task 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 intra-task L2 misses in flight; (s2) It has only inter-task L2 misses in flight; and (s3) It has both inter-task and intra-task L2 misses in flight simultaneously.

We consider a task is not progressing, and hence, it should not be accounted in state (s2). In other words, accounting is stopped when the task experiences an inter-task L2 miss and it cannot overlap its stall with any other intra-task L2 miss. We resume accounting the task when the inter-task L2 miss is resolved or the task experiences an intra-task L2 miss, in which case the task is able to overlap the memory latency of the inter-task L2 miss with at least one intra-task L2 miss. In the state (s3), we do a normal accounting because the inter-task L2 miss overlaps with another intra-task L2 miss.

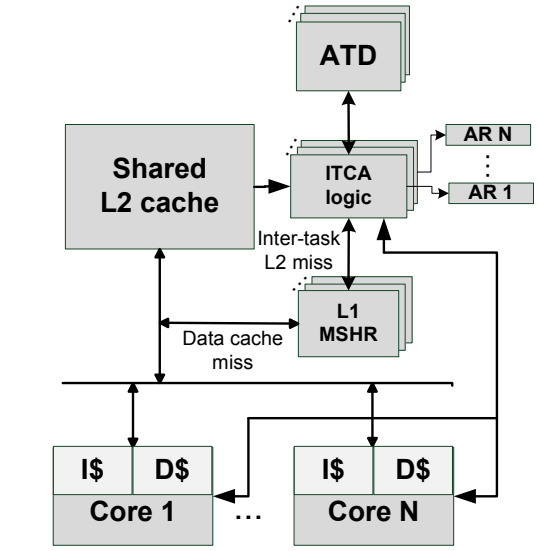
When an inter-task L2 miss becomes the oldest instruction in the Reorder Buffer (ROB) and the ROB is full, the task loses an opportunity to extract more Memory Level Parallelism (MLP). For example, let's assume that there are S instructions between the inter-task L2 miss in the top of the ROB and the next intra-task L2 miss in the ROB. In this situation, if the task had not experienced the inter-task 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 in the top of the ROB is an inter-task L2 miss and the ROB is full. We call this condition (s4).

L2 instruction misses: Another condition in which we stop the accounting of a task, is when the ROB is empty because of an inter-task 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, we have only 1 in flight instruction miss per task at most. Hence, on an inter-task instruction L2 miss we consider that the task is not progressing because of an inter-task conflict, and hence, we stop its accounting.

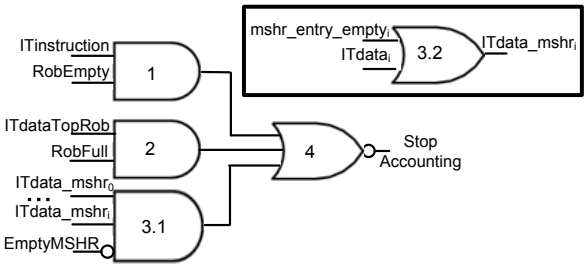
A. Implementation

Figure 4 shows a sketch of the hardware implementation of our proposal. Next, we explain in depth the different parts of our approach.

Detecting inter-task misses: We keep an *Auxiliary Tag Directory* (ATD) [5] for each core (see Figure 4 (a)). 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. While the tag directory of the L2 cache is accessed by all tasks, the ATD of a given task is only accessed by



(a) Baseline processor architecture



(b) Logic to stop accounting

Figure 4: Hardware required for ITCA

the memory operations of that particular task. If the task misses in the L2 cache and hits in its ATD, we know that memory access would have hit in cache if the task had run in isolation [14]. Thus, it is identified as an inter-task L2 miss.

Tracking inter-task misses: We also add one bit called *ITdata* bit in each entry of the Miss Status Hold Register (MSHR). The ITdata bit is set to one when we detect an inter-task 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 cache miss, we have to access the L2 cache. We access the 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 inter-task 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 inter-task L2 cache instruction miss, we stop accounting cycles to this task. For our purpose, we use a bit called *ITinstruction* that indicates whether the task has an inter-task 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 L2 cache instruction miss (gate (1) in Figure 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 L2 cache instruction miss. (2) The ROB is full, in which case *RobFull*=1, and the oldest instruction in the ROB is an inter-task L2 cache data miss which might require one bit per ROB entry (gate (2) in Figure 4 (b) that implements condition (s4)). The signal *RobFull* is already present in most architectures. (3) All the occupied MSHR entries belong to inter-task misses. To compute this, we check whether every entry i of the MSHR is not empty ($mshr\_entry\_empty_i = 0$) and contains an inter-task L2 miss ($ITdata_i$) (gates (3.1) and (3.2) in Figure 4 (b) implement condition (s2)). By making an AND operation of $ITdata\_mshr_i$ and a signal showing whether the entire MSHR is empty, *EmptyMSHR* (3.1), 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.

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. Otherwise, we should consider it as another resource to be tracked by ITCA.

The cycles accounted to each task in each core are saved into a special purpose register per core, *Accounting Register* or *AR* (see Figure 4 (a)), which can be communicated to the OS. This register is a read only register like 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 context switch, the OS reads the Accounting Register (AR_i) of each $task_i$, where AR_i reports the time to account this task. With this information, the OS updates metrics of the system and carries out the scheduling tasks. The OS can alternatively be changed to use the information provided by ITCA similar to [8]. When a task is swapped into a CPU, its associated AR_i is reset. On a task migration, both the ATD and the cache require some time to warm up but we expect this overhead to be low.

IV. Experimental results

A. Experimental environment

Simulator: In order to compare ITCA and the CA, we use the MPsim [9], a highly flexible cycle-accurate simulator that allows us to model CMP architectures. Our baseline configuration is shown in Table I, which represents an architecture

with a 12-stage-deep pipeline. We use 2 processor setups: a dual-core single-thread CMP and a quad-core single-thread CMP.

Workloads: We feed our simulator with traces collected from the whole SPEC CPU 2000 benchmark suite [1] using reference input sets. Each trace contains 300 million instructions, that are selected using SimPoint [10]. From these benchmarks, we generate 2- and 4-task workloads. In each workload, the first task in the tuple is the Principal Task (PT) and the remaining tasks are considered as Secondary Tasks (STs). In every workload, we execute the PT until completion. The other tasks are re-executed until 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 generate all possible 2-task combinations, leading to a total number of 676 workloads. Since it is not feasible to run all 4-task combinations, we classify benchmarks into two groups depending on their memory 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. From these two groups, we generate 8 workload types denoted V_WYZ, where V is the type of the PT and WYZ is the type of the three STs. For example, M_MMI indicates that the PT is memory bound, two of the STs are memory bound and one of the STs is ILP. In total, we use 64 4-task workloads.

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. The ratio

$$\left| 1 - \frac{TA^{CMP}_{PT,I_{PT}}}{TR^{ISOL}_{PT,I_{PT}}} \right| \quad (8)$$

estimates off estimation. The ratio

$$\left| 1 - \frac{TR^{CMP}_{PT,I_{PT}}}{TR^{ISOL}_{PT,I_{PT}}} \right| \quad (9)$$

provides 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*.

B. Accuracy Results

Our results show that for the 2-core CMP configuration, when ITCA takes into account only the conflicts in the L2 cache (gates (1), (3.1) and (3.2) in Figure 4 (b)), it provides a good measure of the progress each task makes with respect to its execution in isolation. While on average, the CA has an off estimation of 9%, ITCA reduces it to 3%. Moreover, ITCA reduces the inaccuracy in the worst five cases: the Avg5WOE metric is 120% for the CA and only 34% for ITCA. In addition to inter-task conflicts in the L2, if ITCA is also aware of when a task loses opportunities of exploiting MLP (gate (2) in Figure 4 (b)), the off estimation reduces down to 1% and the Avg5WOE reduces to 20%.

Table I: Simulator baseline configuration

Number of cores	2 & 4
Fetch policy	ICOUNT 1.8
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, 16 ways, 8 banks, 128B line, 12 cycles access
MSHR	32 entries
Mem latency/BW	300 cycles, 6.4GB per sec. 2 Memory channels

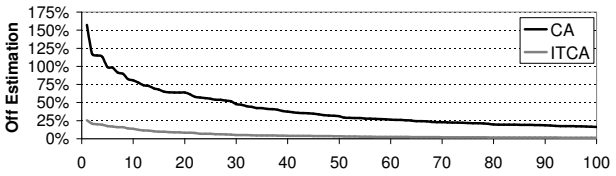
Figure 5 (a) breaks down the results of ITCA and the CA. It shows the 100 workloads with the highest off estimation sorted in descending order. We observe that the CA has higher dispersion than ITCA in the first 50 workloads. This high variability in the CPU accounting may neglect the work of the OS in providing fairness among running tasks. Instead, ITCA provides more stable results: the worst observed off estimation is 25% and this value rapidly converges to zero.

Figure 5 (b) shows the off estimation of ITCA and the CA for the 4-core CMP. In this case, we show the average results of each group as we described in Section IV-A. We observe that the CA obtains the worst results when the PT is ILP and any of the STs is memory bound. 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. Hence, when they run with memory bound tasks, which make an intensive use of the L2 cache, the ILP tasks suffer a lot of inter-task 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 memory bound, 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 memory bound tasks is relatively lower.

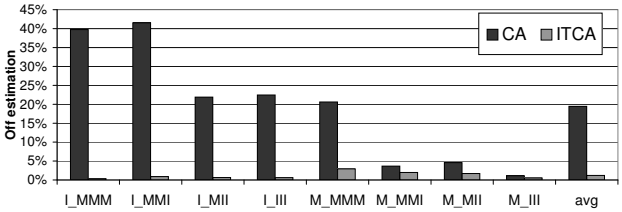
ITCA takes into account inter-task L2 misses and as a consequence it reduces the off estimation of the CA from 19% to 1%. In the worst five cases, the CA has an off estimation of 149% while ITCA has an off estimation of 5%.

C. Reducing the ATD's Overhead

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



(a) 100 highest off estimation (2-core CMP)



(b) Off estimation for 4-core CMP configuration

Figure 5: Off estimation of each approach for 2- and 4-core configurations

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 does not detect some actual inter-task L2 misses.

Second, we use a sampled version of the ATD or sATD [5]. This scheme monitors only a subset of the cache sets (sampled sets). This scheme provides similar results to the ATD in terms of performance [5]. When using sATD with ITCA, for the access to non-sampled sets we cannot determine whether they are inter- or intra-task misses. In this situation, ITCA does not consider this miss in the accounting task.

Figure 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 6 bits of tags (6bitTag-SD2). In this case, we reduce the size of the ATD to 6KB, and increase the average off estimation and Avg5WOE to 4% and 14%, respectively. Recall that in this configuration, the CA leads to an average and Avg5WOE off estimation of 9% and 120%, respectively. Depending on 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 1% and 9%, respectively. For the 4-core setup the results are similar.

In our view, power consumption, rather than area, is a main problem in future processor's design. The ATD is

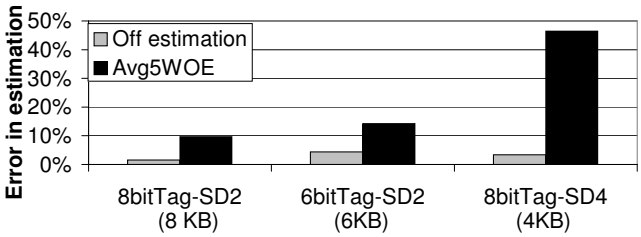


Figure 6: Effect of reducing ATD overhead on accuracy

accessed only when a task misses in the data or instruction cache and moreover, only one entry in the ATD is active at a time, thus its power consumption is low.

V. ITCA and Cache Partitioning Algorithms

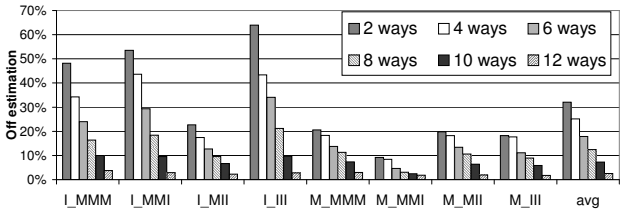
Cache Partitioning Algorithms (CPA) dynamically partition the shared L2 cache among running tasks. CPAs significantly improve metrics like throughput [7], [5], [4], fairness [11] and Quality of Service [7], [12].

An accounting mechanism is also required in the presence of a CPA as tasks suffer slowdowns in their progress since CPA assigns them only a part 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 CPA without changes. The only conceptual difference is that the tasks do not suffer inter-task 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 CPA and our accounting algorithm can make use of it. In such a design, the only hardware cost of ITCA is the logic shown in Figure 4 (b).

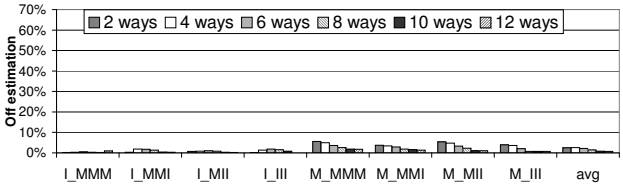
In the previous sections, ITCA was evaluated on a CMP with a shared L2 cache with Least Recently Used (LRU) as replacement policy. The LRU scheme tends to give more space to the tasks that access more frequently to the cache hierarchy. Next, we evaluate the accuracy of ITCA when using a partitioned cache with two different schemes: static partitioning and dynamic (MinMisses [5]). The static partitioning scheme determines the amount of cache size allocated to each task at the beginning of its execution. Instead, MinMisses dynamically changes the partition to adapt to the varying demands of competing tasks. This algorithm attempts to minimize the total number of misses among all tasks sharing the cache.

For this study we compare the off estimations of the CA and ITCA on a 4-core configuration. We use the same workload groups explained in Section IV-A.

Figure 7 (a) shows the average off estimation of the CA in combination with the static CPA as we increase the number of ways assigned to the PT. We observe that the accuracy improves as the number of ways given to the PT increases. This is intuitive because, as the PT receives more cache



(a) Off estimation of the CA



(b) Off estimation of ITCA

Figure 7: Off estimation of the CA and ITCA for 4-core CMP using a static CPA

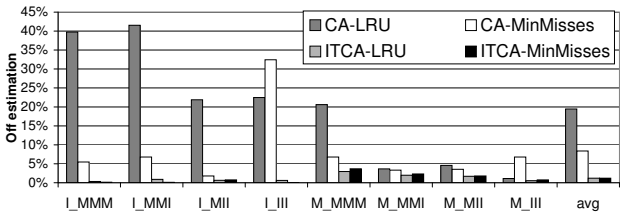


Figure 8: Off estimation of the CA and ITCA for 4-core CMP with dynamic CPA using different replacement policies

space, it suffers less inter-task L2 cache misses and hence, its execution time is closer to its execution time in isolation. The same behavior is observed for CPU and memory bound tasks. However, memory bound tasks require more cache space to reach their maximum performance. As a result, the off estimation reduction is slower compared to CPU bound tasks. When the PT has roughly all the L2 cache, the off estimation is significantly reduced and the time accounted to the PT is similar to its execution time in isolation.

Figure 7 (b) shows the same results for ITCA. We observe that ITCA reduces the off estimation to less than 6%, in all groups (between 0.9% and 2.8% on average). Instead, the CA presents an off estimation up to 63% in group I_III.

Figure 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 IV-B. 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 1% in average, as we observed in Section IV-B.

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 that the PT suffers a lot of inter-task misses, increasing the off estimation. As a consequence, in some groups, the off estimation is high (I_III and M_III), reaching a 32% off estimation in group I_III.

ITCA-MinMisses provides much more stable results than CA-MinMisses. ITCA-MinMisses reduces the average off estimation of the CA-MinMisses from 9% to 1%. In comparison with CA-MinMisses, ITCA-MinMisses consistently reduces the off estimation to less than 3% in all groups (including I_III and M_III).

To sum up, the combination of ITCA either with static or 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.

VI. Other Issues regarding CPU Accounting

A. Other Proposals providing Fairness

Several hardware approaches deal with the problem of providing fairness in multicore architectures. Although, fairness is a desirable characteristic of a system, next we show that it cannot be used to provide an accurate CPU accounting. There are two main flavors of fairness.

First, it is assumed that an architecture is fair when it gives the *same amount of resources* to each running task. However, ensuring a fixed amount of resources to a task [13], [7], [14], [15], does not translate into a CPU utilization that can be computed for that task. This is due to fact that the relation between the amount of resources assigned to a task and its performance can be different for each task.

The second flavor of fairness considers 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 tasks X and Y. The system is said to be fair if in a given period of time, the progress made by X and Y is the same, $P_X = P_Y$. However, the fact that $P_X = P_Y$ does not provide a quantitative value that can be provided to OS, so that it can account CPU time to each task. In other words, to know that $P_X = P_Y$ does not provide any information about CPU accounting since P_X can be any value lower than 1. In Figure 9, we show the progress of the PT and the fairness of four different pairs of tasks measured as $(1-((|P_X-P_{AVG}|+|P_Y-P_{AVG}|)/2))$, where P_{AVG} is the average progress made by 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 (*galgel*+applu and *ammp*+ammp), the PT (task in italics) does the same progress while the fairness is

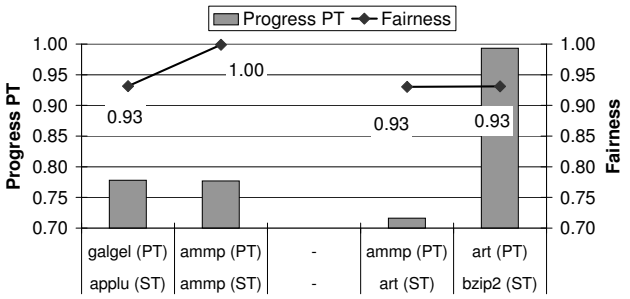


Figure 9: Fairness and progress of the PT for four different workloads

different in the two workloads. For the two workloads on the right (*ammp*+art and *art*+bzip2), both workloads present the same fairness while the progress done by the PT is different.

In conclusion, multicore systems that provide fairness do not necessarily provide accurate CPU accounting. Therefore, these systems require a CPU accounting mechanism as well.

B. Alternative Accounting Mechanisms

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. 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 utilization. In order to support this claim, we try to provide an accurate measure of the CPU utilization 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 CMP, TR_{CMP}, weighted by the percentage of instructions this task X executes, I_X, with respect to all the instructions executed in the workload. We call this approach, *Performance Counter Instruction-Based* (*PCIB*)

For example, let's assume we execute a workload composed of two tasks X and Y in a multicore 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 the 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 a CMP processor with N cores. That is, $TA_X^{CMP} = (1/N)TR_X^{CMP}$. In this approach, we assume that each task receives an even part of the resources of the processor and that it makes $1/N$ of the progress it would do if run in isolation.

Results: Our results show that for the 2-core configuration, PCIB and sCA report an average off estimation of 47% and 46%, respectively. 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 70%. The Avg5WOE is 93% and 75% 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's assume that we run a memory bound task as PT and an ILP bound task as ST. In this situation, the ST executes much more instructions than the PT, so the PT is accounted a very small fraction of the time it is running. However, in reality the PT is making almost the same progress as in isolation since it is not suffering inter-task cache misses. This introduces a significant error in the CPU accounting of the PCIB scheme.

C. Multithreaded Tasks

In this paper, we have evaluated ITCA with multiprogrammed workloads (workloads in which each task is single threaded), but we expect that ITCA will also work with minimal changes for multithreaded tasks. The interaction between threads in a parallel task can be *positive* when, for example, one thread prefetches data for another thread. This behavior is intrinsic to the task, and hence to its execution in isolation. When one multithreaded task runs with other tasks it may suffer *negative* interaction, i.e. it may suffer inter-task 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 inter-task 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 inter-task miss and stop the accounting if necessary. No other changes are required in the ITCA implementation.

With ITCA, the CMP processor reports an accounting for each thread of a multithreaded task through the respective Accounting Register (AR) in each core. It is the responsibility of the accounting mechanism done at the OS level, like [8], to obtain a single accounting figure from the accounting done to each task. How to get this figure is out of the scope of this paper.

Notice that the hardware accounting mechanism (ITCA) does not have to be aware of the synchronization among

threads of a multithreading task. For example, if a thread is spinning on a lock and even if the multithreading task is not progressing during that time, the thread is using the core, so it is accounted processor time. It is the responsibility of the programmer to reduce waiting times when acquiring locks or to release the processor until the lock is freed, in which case the accounting for that thread stops.

VII. 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 [16], [17], 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) [18]. 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) [19], 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 [20] introduces a similar mechanism to PURR. The main difference with PURR is that the target SMT processor in [20] 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 [20] 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 decodes. For example, if in a given cycle the first task decodes 5 instructions and the second 2, the first task is charged 5/7 and the second 2/7.

A recent paper [21] proposes a new cycle accounting architecture for SMT processors based on estimating the CPI stack of each running task [22]. This proposal tracks fifteen different components of the CPI stack with a dedicated hardware. It also uses an ATD per hardware thread to scale the CPI components related to the cache hierarchy, which implies using floating point multiplication operations. 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.

In the operating system domain, the most similar work to our proposal is [8]. In [8], 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 co-runners 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. Our work is different from [8] in 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 in [8] can be used on top of our mechanism to compensate the time quantum of tasks to meet their expected performance.

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 mentioned above for SMT architectures with our ITCA proposal for CMP processors.

VIII. Conclusions

CMP architectures introduce complexities in the CPU accounting 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. This accounting inaccuracy may affect several key elements of the system like the OS task scheduling or the charging mechanism in data centers. We present a hardware support for a new accounting mechanism called *Inter-Task Conflict Aware* (ITCA) accounting that improves the accuracy of the CA. In a 2- and 4-core CMP architecture, ITCA reduces the off estimation down to 1% while the CA presents a 9% and 19%, respectively. We also have shown that multicore systems that provide *fairness* do not necessarily provide accurate CPU accounting. As a consequence, these systems require a CPU accounting scheme to provide accurate measurements of the CPU utilization.

Finally, the combination of ITCA with static and dynamic CPAs significantly reduces the off estimation with respect to the CA. Furthermore, ITCA leverages the ATDs already present in many cache partitioning algorithms (like the MinMisses scheme) with nearly no extra hardware addition motivating the use of ITCA and CPAs simultaneously.

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 AP-2005-3318, 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 of Gran Canaria for his help setting up the Intel Xeon Quad-Core processor, and to the reviewers for their valuable comments.

REFERENCES

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

[2] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *in IEEE Computer*, vol. 30, no. 9, 1997.

[3] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "CPU Accounting in CMP Processors," *in IEEE Computer Architecture Letters*, vol. 8, no. 1, 2009.

[4] G. E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," in *HPCA*, Boston, Massachusettes, USA, 2002.

[5] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, Orlando, Florida, USA, 2006.

[6] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable Performance in SMT Processors: Synergy between the OS and SMTs," *in IEEE Trans. Computers*, vol. 55, no. 7, 2006.

[7] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt, "QoS policies and architecture for cache/memory in CMP platforms," in *SIGMETRICS*, San Diego, California, USA, 2007.

[8] A. Fedorova, M. Seltzer, and M. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *IEEE PACT*, Brasov, Romania, 2007.

[9] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero, "The MPsim Simulation Tool," in UPC, Tech. Rep. UPC-DAC-RR-CAP-2009-15, 2009.

[10] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," in *IEEE PACT*, Barcelona, Spain, 2001.

[11] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *IEEE PACT*, Antibes Juan-les-Pins, France, 2004.

[12] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero, "FlexDCP: a QoS framework for CMP architectures," *in SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, 2009.

[13] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Architectural Support for Real-Time Task Scheduling in SMT Systems," in *CASES*, San Francisco, California, USA, 2005.

[14] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual Private Caches," in *ISCA*, San Diego, California, USA, 2007.

[15] S. E. Raasch and S. K. Reinhardt, "The Impact of Resource Partitioning on SMT Processors," in *IEEE PACT*, New Orleans, LA, USA, 2003.

[16] D. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *ISCA*, Santa Margherita Ligure, Italy, 1995.

[17] M. J. Serrano, R. Wood, and M. Nemirovsky, "A Study of Multistreamed Superscalar Processors," in UCSB, Tech. Rep. #93-05, 1993.

[18] P. Mackerras, T. S. Mathews, and R. C. Swanberg, "Operating system exploitation of the POWER5 system," *in IBM J. Res. Dev.*, vol. 49, no. 4/5, 2005.

[19] 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," *in IBM J. Res. Dev.*, vol. 51, no. 6, 2007.

[20] R. L. Arndt, B. Sinharoy, S. B. Swaney, and K. L. Ward, "Method and apparatus for frequency independent processor utilization recording register in a simultaneously multithreaded processor," *US PATENT 20060173665*, 2006.

[21] S. Eyerman and L. Eeckhout, "Per-Thread Cycle Accounting in SMT Processors," in *ASPLOS*, Washington, DC, USA, 2009.

[22] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *ASPLOS*, San Jose, CA, USA, 2006.