CPU Accounting in CMP Processors

Carlos Luque[1], Miquel Moreto[1], Francisco J. Cazorla[2], Roberto Gioiosa[3], Alper Buyuktosunoglu[3], Mateo Valero[1,2]
[1]Universitat Politècnica de Catalunya [2]Barcelona Supercomputing Center [3]IBM T. J. Watson Research Center

Abstract— Chip-MultiProcessors (CMP) introduce complexities when accounting CPU utilization to processes because the progress done by a process during an interval of time highly depends on the activity of the other processes it is co-scheduled with. We propose a new hardware 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 could lead to a 12% 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.

1. 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 machine while, in fact, the OS shares hardware resources among the users. Hardware resources can be shared in two possible ways: *temporarily* and *spatially*. Hardware resources are time shared between users when each process 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 process makes use of a limited amount of resources, like the cache memory or the I/O bandwidth.

Even if the user application perceives to be alone in the system, its execution time is affected by the amount of hardware resources shared with the other running applications and for how long. However, *the time accounted to that application should always be the same regardless of the workload*[1] *in which it is executed, i.e., regardless of how many processes are sharing the hardware resources at any given time*. Unix-like systems differentiate the real execution time and the time a process actually is running on a CPU. Commands like `time` or `top` provide three outcomes: `real`, `user` and `sys`. `real` is the total elapsed (wall clock) time used by the process; `user` is the time the process used directly the CPU; and `sys` is the time spent in kernel mode on behalf of the process. In these systems, `sys+user` time is the execution time accounted to the process.

Figure 1 shows the total (`real`) and the accounted execution time (`sys+user`) of the `459.GemsFDTD` (or simply `gems`) SPEC CPU 2006 benchmark when it runs in different workloads. The time results in this figure are normalized to the real execution time of `gems` when it runs in isolation (16.52 minutes). For this experiment, we use Linux 2.6.24 on an Intel Core 2 Duo 1.6 GHz machine, which has a dual-core chip in which each core is single threaded (though the general trends drawn from Figure 1 apply to all current CMPs). We isolate one core to emulate a uniprocessor system (single thread or ST mode): the OS activity was bound to the first core, leaving the second core as isolated as possible from noise. When `gems` runs together with other processes in the same core, its `real` execution time increases up to around 2x due to context switches between all running processes. Nevertheless, `gems` is accounted roughly the same time (grey triangles) which is the time the process actually uses the CPU. Processes 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 `gems`'s total execution time increases depending on the other application it is co-scheduled with, the time accounted to `gems` is always the same. In uniprocessor systems, each running process 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 SMP[2] systems, as the amount of shared resources is limited and the major task of the OS scheduler is to time share the CPUs between the runnable processes.

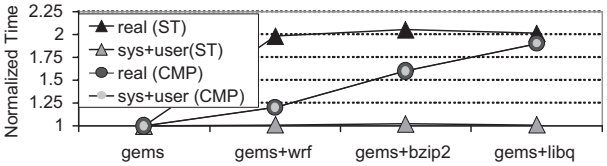


Fig. 1. Total (`real`) and accounted (`sys+user`) time of `gems` in different workloads and processor (Intel Core 2 Duo) configurations

However, processors with shared resources, like CMPs, make CPU accounting more complex because the progress of a process depends on the activity of the other processes running at the same time. Current OSs still use the CA for multithreaded processors, which can lead to inaccuracy for the time accounted to each process. To show this inaccuracy, in a second experiment, we use both cores of the Intel Core 2 Duo processor. Next, we execute `gems` with several workloads as shown by the x-axis in Figure 1. In this case, given that the number of processes running is equal or less than the number of virtual CPUs (cores) in the system, processes suffer no time sharing and `real` time is roughly the same as `sys+user`. In Figure 1, the grey circles show a variance up to 1.9x in the time `gems` is accounted depending on the workload in which it runs. This means that (at least with current known open source OSs like Linux) *a process running on a CMP processor may be accounted differently according to the other processes 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 processes it is co-scheduled with.

CPU accounting affects several key componets of a computing system: For example, if the OS scheduler is not able to properly account the CPU utilization of each process, the OS scheduling algorithm will fail to maintain fairness between processes. As a consequence the scheduling algorithm cannot guarantee that a process progresses with its work as expected. In data centers, customers are charged according to the utilization of the CPU they use. Hence, an accurate accounting is also critical in this scenario.

In this paper, we make for the first time a comprehensive analysis of the CPU accounting accuracy of the CA that, as far as we know, is the only accounting mechanism for CMPs. Next, we propose a hardware mechanism, *Inter-Thread Conflict-Aware* (ITCA) accounting, that improves the accuracy of the CA for CMPs. In a 2-core CMP architecture, ITCA reduces the inaccuracy to 1% (20% in the worst five cases), while the CA presents an inaccuracy of 9% (120% in the worst five cases). In a 4-core CMP processor, ITCA leads to an inaccuracy less than 1% (9% in the worst five cases), while the CA shows an inaccuracy of 12% (124% in the worst five cases).

2. Formalizing the Problem

Currently, the OS perceives the different cores in a CMP as multiple independent virtual CPUs. With the CA the OS does not consider the interaction between processes caused by shared resources. However, the time running on a virtual CPU is not an

Manuscript submitted: 04-Feb-2009. Manuscript accepted: 18-Mar-2009. Final manuscript received: 25-Mar-2009

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

[2]Symmetric Multi-Processors have single thread, single core chips. SMPs share off-chip resources like the memory bandwith or the I/O channels. We consider those resources less critical and only focus on on-chip resources.

DOI 10.1109/L-CA.2009.3 1556-6056/09/$25.00 © 2009 Published by the IEEE Computer Society

accurate measure of the amount of CPU resources the process has received. The time to account to a process in a CMP processor does not only depend on the time that process is scheduled onto a CPU, but also on the progress, it makes during that time. In our view, in CMPs, we have to maintain the same principle that rules today in SMP and uniprocessor systems: Let's assume that if a process X runs for a period of time in a CMP, TR^{CMP}_{X,I_X}, in which it executes I_X instructions. The actual time to account this process TA^{CMP}_{X,I_X}, is the time it would take this process to execute in isolation these I_X instructions in the same architecture, denoted TR^{ST}_{X,I_X}. This would make the CPU accounting independent from the rest of the workload.

With the CA, the time accounted to X in a uniprocessor system, TA^{CA}_{X,I_X}, can be expressed as $TA^{CA}_{X,I_X} = TR^{ST}_{X,I_X}$. Next, let's assume that we run X in a workload on a multithreaded architecture for TR^{CMP}_{X,I_X} units of time[3], executing the same I_X instructions. The relative progress that process X has in this interval of time can be expressed as $P^{CMP}_{X,I_X} = TR^{ST}_{X,I_X}/TR^{CMP}_{X,I_X}$. The relative progress can also be expressed as $P^{CMP}_{X,I_X} = IPC^{CMP}_{X,I_X}/IPC^{ST}_{X,I_X}$ in which IPC^{CMP}_{X,I_X} and IPC^{ST}_{X,I_X} are the IPC of process X when executing the same I_X instructions in the CMP and running alone, respectively. Then, $TA^{CMP}_{X,I_X} = TR^{CMP}_{X,I_X} \cdot P^{CMP}_{X,I_X} = TR^{ST}_{X,I_X}$ that fulfills our principle of workload-independent accounting.

The main issue to address is how to determine dynamically (while a process X is simultaneously running with other processes) at the end of each context switch, the time (or IPC) it would take X to execute the same instructions if it had been alone in the system. An intuitive solution is to provide hardware mechanisms to determine the IPC in isolation of each process in a workload by periodically running each process in isolation [3], [6]. However, as the number of processes simultaneously executing in a multithreaded processor increases to dozens or even hundreds, this solution will not scale, as the number of isolation phases increases linearly with the number of processes in the workload. As a consequence, the time the processor runs in multithreaded mode is reduced, affecting the system performance.

3. Inter-Thread Conflict-Aware (ITCA) Accounting

In this paper, we refer to *inter-thread* resource conflicts to those resource conflicts that a thread[4] suffers due to the interference of the other threads running at the same time. For example, a given thread X suffers an inter-thread L2 cache miss when it accesses a line that was evicted by another thread, but would have been in cache, if X had run in isolation. Likewise, *intra-thread* resource conflicts denote those resource conflicts that a thread suffers even if it runs in isolation.

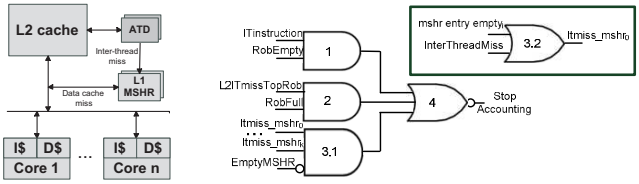
The CA leads to over estimation in CMPs (meaning that they are accounted more time than the time they require to make the same progress when running in isolation), since for a process X, we have $TA^{CA}_{X,I_X} = TR^{CMP}_{X,I_X} \geq TR^{ST}_{X,I_X}$. In our baseline processor setup (see Figure 2(a)). The data and instruction caches are private to each core and the L2 is shared. The main source of over estimation are inter-thread conflicts and, in particular, *inter-thread L2 misses*, which delay the threads with respect to their isolated execution.

The idea of our proposal is to account a thread for only those cycles in which the thread is not stalled due to an inter-thread L2 cache miss. That is, we want to account a thread only when it is progressing or it is stalled because of an intra-thread L2 miss.

L2 data misses: In order to do so, we consider a thread is in one of the following states: (s1) It has no L2 cache misses or it has only intra-thread L2 misses in flight; (s2) It has only inter-thread L2 misses in flight; and (s3) It has both inter-thread and intra-thread L2 misses in flight simultaneously. We consider that a thread is not progressing

[3]In Unix-like systems, we have $TR^{CMP}_{X,I_X} = \texttt{user+sys}$.

[4]In the paper, we interchangeably use the terms of process and thread



(a) Baseline processor arch. (b) Logic to determine the accounting

Fig. 2. Hardware required for the ITCA accounting approach

and hence should not be accounted in state (s2). That is, we stop accounting that thread when the thread experiences an inter-thread miss and it cannot overlap its delay with any other intra-thread miss. We resume accounting for the thread when the inter-thread miss is resolved or the thread experiences an intra-thread miss, in which case the thread is able to overlap the memory latency of the inter-thread miss with at least one intra-thread miss.

In the state (s3), we do a normal accounting because the inter-thread miss overlaps with another intra-thread miss. However, when the inter-thread miss becomes the oldest instruction in the Reorder Buffer (ROB) and the ROB is full, the thread loses an opportunity to extract more Memory Level Parallelism (MLP). That is, let's assume that there are Y instructions between the inter-thread L2 miss in the top of the ROB and the next intra-thread L2 miss in the ROB. In this situation, if the thread had not experienced the inter-thread L2 miss it would have executed the Y instructions after the last instruction currently in the ROB. Any L2 miss in these Y 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 thread while the instruction in the top of the ROB is an inter-thread L2 miss and the ROB is full. We call this situation (s4).

L2 instruction misses: Another situation in which we stop the accounting of a thread, is when the ROB is empty because of an inter-thread 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 thread. Hence, on an inter-thread instruction L2 miss we consider that the thread is not progressing because of an inter-thread conflict, and hence, we stop its accounting.

3.1. Implementation

Detecting inter-thread misses: We keep an Auxiliary Tag Directory (ATD) [10] for each core (see Figure 2(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 thread in isolation (ST mode). While the tag directory of the L2 cache is accessed by all threads, the ATD of a given thread is only accessed by the memory operations of that particular thread. If the thread misses in the L2 cache and hits in its ATD, we know that memory access would hit in cache if the thread was running in isolation. Thus, it is identified as an inter-thread miss.

Tracking inter-thread misses: We also add one bit (inter-thread bit or IT bit) in each entry of the Miss Status Hold Register (MSHR), which is set to 0 when the entry is allocated. 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 thread 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-thread L2 cache conflict and the IT 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-thread instruction cache miss, we stop accounting cycles to this thread. For our purpose, we

use a bit, *ITinstruction*, that indicates whether the thread has an inter-thread L2 cache instruction miss or not.

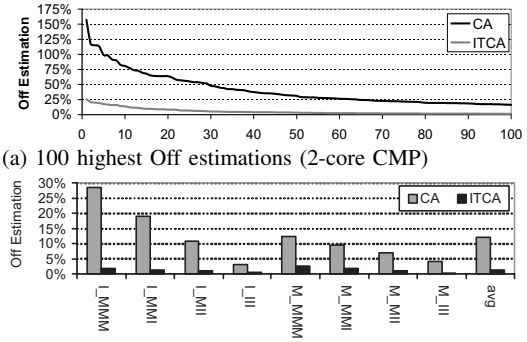
Accounting CPU time: We stop the accounting of a given thread when: First, the ROB is empty because of a L2 cache instruction miss (gate (1) in Figure 2(b) that implements situation (s5)). *RobEmpty* is a signal that is already present in most architectures, while *ITinstruction* indicates whether or not a thread has an L2 cache instruction miss. Second, the ROB is full (signal already present in most architectures) and the oldest instruction in the ROB is a data inter-thread L2 miss, which can be implementing adding 1 bit per ROB entry (gate (2) in Figure 2(b) that implements situation (s4)). Third, when all the occupied MSHR entries belong to inter-thread misses. To compute this, we check for every entry k of the MSHR if an entry is not empty ($mshr\_entry\_empty_i = 0$) and contains an inter-thread miss ($InterThreadMiss_i$) (gates (3.1) and (3.2) in Figure 2(b) that implement situation (s2)). By making an AND operation with the output for each MSHR entry and a signal showing whether the entire MSHR is empty, $EmptyMSHR$ (3.1), we determine if we have to stop the accounting of the thread. Finally, if any of the gates (1), (2) or (3.1) returns 1, we stop the accounting.

To summarize, in a 2-core CMP, ITCA accounts every spent cycle in three possible ways: (1) Each thread is accounted for the cycle when both threads progress (the cycle is accounted twice, one for each thread). (2) Only one thread is progressing and the cycle is accounted only to it. (3) The cycle is not accounted to any thread when none is progressing. In our processor setup, the memory bandwidth is not identified as a main source of interaction between threads. Otherwise, we should consider it as an other resource to be tracked by ITCA. The CPU accounting done to each thread in each core can be communicated to the OS by a special purpose register that counts cycles, like the Time Stamp Counter in Intel architectures.

4. Experimental Results

4.1. Experimental Environment

We use MPsim [1], a trace driven CMP simulator to model two processor setups: a dual-core and a quad-core CMP. Each core is single threaded, has a 12-stage-deep pipeline and can fetch up to 8 instructions each cycle (ICOUNT 1.8). Each core has 6 integer (I), 3 floating point (FP), and 4 load/store functional units; 64-entry integer, load/store, and FP instruction queues; 512-entry reorder buffer and 196 I/FP physical registers. We use a two-level cache hierarchy with 128B lines with a separate 16KB, 4-way instruction cache and a 64KB, 4-way data cache and a unified 2MB, 16-way L2 cache that is shared among all cores. The latency from L1 to L2 is 12 cycles, and from L2 to memory 300 cycles. We feed our simulator with traces collected from the whole SPEC CPU 2000 benchmark suite using the reference input set. Each trace contains 300 million instructions, selected using SimPoint [11]. From these benchmarks, we generate 2- and 4-thread workloads. In each workload, the first thread in the tuple is the Principal Thread (PTh) and the remaining threads are considered Secondary Threads (STHs). In every workload, we execute the PTh until completion. The other threads are re-executed until PTh completes. We characterize the results of our proposal based on the type of the PTh and STHs. We generate all possible 2-thread combinations, leading to a total number of 676 workloads. Running all 4-thread combinations is infeasible as the number of combinations is too high. Hence, we classify benchmarks in two groups depending on their memory behavior. Benchmarks in the memory group (denoted M) are those presenting a bad L2 cache behavior (mainly `art`, `equake`, `lucas`, `mcf` and `swim`), while benchmarks in the ILP group (denoted I) have a low L2 cache miss rate (mainly `bzip2`, `crafty`, `eon`, `gcc` and `gzip`). From these two groups, we generate 8 workload types denoted V_WYZ, where



(a) 100 highest Off estimations (2-core CMP)

(b) Off estimation in the 4-core CMP configuration

Fig. 3. Off estimation of each approach for 2- and 4-core configurations.

V is the type of the PTh and WYZ the type of the three STHs, e.g. M_MMI indicates that the PTh and two of the STHs are memory bound, while one STh is ILP.

As the main metric, we measure how off is the estimation done by an accounting approach for the PTh ($TA^{CMP}_{PTh,I_{PTh}}$), which allows us to break down the results according to the type of PTh (ILP/MEM) and type of the STHs, from the actual time it should be accounted ($TR^{ST}_{PTh,I_{PTh}}$). We call *off estimation* to the ratio $\left|1-(TA^{CMP}_{PTh,I_{PTh}}/TR^{ST}_{PTh,I_{PTh}})\right|$. This ratio is $\left|1-(TR^{CMP}_{PTh,I_{PTh}}/TR^{ST}_{PTh,I_{PTh}})\right|$ for the CA. For each accounting policy, we also report the average off estimation of the five workloads with the worst off estimation, denoted *Avg5WOE*.

4.2. Accuracy Results in a CMP processor

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 2(b)), it provides a good measure of the progress each process makes with respect to its execution in isolation. While on average, the CA has an off estimation of 9%, ITCA reduces it to 3%. More importantly, ITCA reduces the inaccuracy in the worst five cases: the Avg5WOE metric is 120% for the CA and only 34% for ITCA. If in addition to inter-thread conflicts in the L2, ITCA is also aware of when a thread loses opportunities of exploiting MLP (gate (2) in Figure 2(b)), the off estimation reduces down to 1% and the Avg5WOE reduces to 20%.

Figure 3(a) breaks down the results of ITCA and CA and 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 of providing fairness among running processes. ITCA instead provides more stable results: the worst observed off estimation is 25% and this value rapidly converges.

Figure 3(b) shows the off estimation of ITCA and the CA proposal for the 4-core setup. In this case, we show the average results of each group as we presented in the experimental environment section. The CA obtains the worst results when the PTh thread has high ILP and any of the STHs is memory bound. In this case, the PTh suffers a lot of inter-thread conflicts that are not taken into account by the CA. In those cases, the ITCA approach reduces the off estimation of the CA from 12% to 1%. In the five worst cases, the CA has an off estimation of 124% while ITCA has an off estimation of 12%.

ATD: Our baseline ATD has a size of 30KB (15-bit tag, 1024 sets, 16 ways per set). In order to reduce the area requirements, we also implement two simplified versions of the ATD. In the first proposal, in the ATD, we save a subset of the address' tag bits for each memory operation. This proposal introduces false hits when the subset of the tag of the memory operation coincide with the bits of the ATD, but the other bits of the tag (not saved in the ATD) are different. The second version is the *sampled ATD*, that monitors a subset of the cache sets and has been shown to provide similar results to the ATD [10].

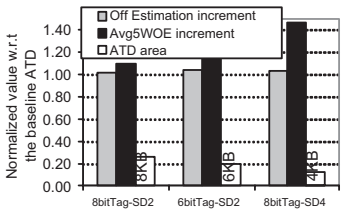


Fig.4 Effect on accuracy of different ATD configurations

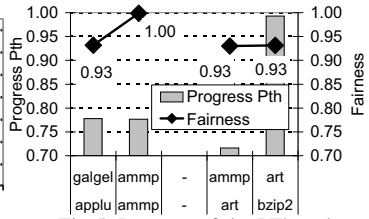


Fig 5. Progress of the PTh and fairness of four pairs of benchmarks

Figure 4 shows the area reduction and accuracy degradation of the simplified versions of the ATD with respect to our baseline ATD. A good tradeoff 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 the Avg5WOE to 4% and 37%, respectively. Recall that in this configuration, the CA leads to an average off estimation and Avg5WOE of 9% and 120%, respectively. Depending on the hardware budget available, different tradeoffs are possible. For example, if 8KB of area can be afforded by core, we can reduce the average off estimation and Avg5WOE to 1% and 9%, respectively. For the 4-core setup the results are similar.

Cache partitioning algorithms: ATDs are also used in *cache partitioning algorithms* (CPA). CPAs dynamically partition the shared L2 cache among running threads and significantly improve metrics like throughput [10] and fairness [8]. An accounting mechanism is required in the presence of a CPA as running processes suffer slowdowns in their progress since the CPA assigns them only a part of the L2 cache. The cache partition a process receives changes dynamically, so the progress of the process (and hence the CPU time to account to it) also changes. Our ITCA proposal can be applied to systems with a CPA with *no changes*. The only conceptual difference is that the running processes do not suffer inter-thread conflicts as each process has a separate partition of the cache. However, we consider that a process is not progressing due to the CPA when it suffers a miss in the L2 cache and a hit in its ATD. Notice that, in systems with CPAs, the ATD is already present and our accounting algorithm can make use of it. In such case the only hardware cost of ITCA is the logic shown in Figure 2(b). Moreover, due to the wide use of the ATD, some authors have already proposed versions of the ATD that require dozens of bytes per thread [7], rather than thousands as it is the case in our baseline architecture. We plan to use this new ATD scheme as a part of the ITCA proposal.

Hardware proposals to provide fairness: Several hardware approaches deal with the problem of *fairness* in multithreaded architectures. However, even if fairness is a desirable characteristic for a system, it cannot be used to provide 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 thread. However, ensuring a fixed amount of resources to a thread, does not translate into a CPU utilization that can be computed to that thread because the relation between the amount of resources assigned to a thread and its performance is different for each thread [2], [6], [9]. Second, some proposals consider that an architecture is fair when all threads running on that architecture make the *same progress*. For example, assume a 2-core CMP with processes A and B. The system is said to be fair if in a given period of time, $P_A = P_B$. However, this approach does not provide a quantitative value that can be given to the OS to account CPU time to each process. That is, knowing that $P_A = P_B$ does not provide any information about CPU accounting as P_A can be any value lower than 1. Figure 5 shows the progress of the PTh and the fairness $(1-(|P_A-P_{AVG}|+|P_B-P_{AVG}|)/2)$, where P_{AVG} is the average progress made by A and B, of four different pairs of benchmarks. We observe that, in the two workloads in the left (`galgel+applu` and `ammp+ammp`), the PTh does the same progress while the fairness is different. In the two workloads on the right (`ammp+art` and `art+bzip2`), both workloads present the same fairness while the progress done by the PTh is different.

5. Related Work

In [5], whenever the OS detects that a process does not make the progress it is supposed to do, the OS increases its time quantum, giving more temporal resources to it and, thus, allowing it to catch its expected performance. The proposed solution is divided in two components: During a *sample* phase the OS runs a process with all the possible co-runners and uses a model to estimate the process' *Fair IPC*. During the *scheduling* phase, the OS scheduler adapts the time quantum of the application in order to provide good performance isolation. In our proposal instead we do not use a model to estimate the isolated performance of a process but we provide hardware support to the OS in order to accurately account each process for the progress it makes. Once the accurate accounting is available, the OS scheduler proposed in [5] can be used on top of our mechanism to compensate the quantum.

Other accounting approaches have been proposed for SMTs [4]. Combining our proposal with these solutions is left for future work.

6. Conclusions

CMPs complicate the CPU accounting because the execution progress done by a process during a given interval of time varies depending on the activity of the other co-scheduled processes. 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 presented a hardware support for a new accounting mechanism called *Inter-Thread 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 12%, respectively.

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 would like to thank Pradip Bose and Chen-Yong Cher from IBM for their technical support.

References

[1] Acosta et al. The MPsim simulation tool. Technical Report UPC-DAC-RR-2009-7, 2009.
[2] Cazorla et al. Architectural support for real-time task scheduling in SMT systems. *In CASES*, 2005.
[3] Cazorla et al. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Trans. Computers*, 2006.
[4] Eyerman et al. Per-thread cycle accounting in SMT processors. *In ASPLOS*, 2009.
[5] Fedorova et al. Improving performance isolation on chip multiprocessors via an operating system scheduler. *In PACT*, 2007.
[6] Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.
[7] Jaleel et al. Adaptive insertion policies for managing shared caches. In *PACT*, 2008.
[8] Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.
[9] Nesbit et al. Virtual private caches. *In ISCA*, 2007.
[10] Qureshi et al. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.
[11] Sherwood et al. Basic block distribution analysis to find periodic behavior and simulation points in applications. *In PACT*, 2001.