RESEARCH-ARTICLE

Fair CPU time accounting in CMP+SMT processors

CARLOS LUQUE, Barcelona Supercomputing Center, Barcelona, Barcelona, Spain

MIQUEL MORETÓ, Barcelona Supercomputing Center, Barcelona, Barcelona, Spain

F. J. CAZORLA, Artificial Intelligence Research Institute, Cerdanyola del Valles, Barcelona, Spain

M. VALERO, Barcelona Supercomputing Center, Barcelona, Barcelona, Spain

Published: 20 January 2013
Accepted: 01 November 2012
Revised: 01 November 2012
Received: 01 June 2012

Fair CPU Time Accounting in CMP+SMT Processors

CARLOS LUQUE, Universistat Politècnica de Catalunya, and Barcelona Supercomputing Center
MIQUEL MORETO, International Computer Science Institute, Universistat Politècnica de Catalunya, and Barcelona Supercomputing Center
FRANCISCO J. CAZORLA, Barcelona Supercomputing Center, and Spanish National Research Council (IIIA-CSIC)
MATEO VALERO, Universistat Politècnica de Catalunya, and Barcelona Supercomputing Center

Processor architectures combining several paradigms of Thread-Level Parallelism (TLP), such as CMP processors in which each core is SMT, are becoming more and more popular as a way to improve performance at a moderate cost. However, the complex interaction between running tasks in hardware shared resources in multi-TLP architectures introduces complexities when accounting CPU time (or CPU utilization) to tasks. The CPU utilization accounted to a task depends on both the time it runs in the processor and the amount of processor hardware resources it receives. Deploying systems with accurate CPU accounting mechanisms is necessary to increase fairness. Moreover, it will allow users to be fairly charged on a shared data center, facilitating server consolidation in future systems.

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

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General

General Terms: Design, Performance, Measurement

Additional Key Words and Phrases: CPU accounting, multicore/multithreaded processors, progress, fairness, slowdown

ACM Reference Format:
Luque, C., Moreto, M., Cazorla, F. J., and Valero, M. 2013. Fair cpu time accounting in cmp+smt processors. ACM Trans. Architec. Code Optim. 9, 4, Article 50 (January 2013), 25 pages.
DOI = 10.1145/2400682.2400709 http://doi.acm.org/10.1145/2400682.2400709

This work was supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625. C. Luque held the FPI grant BES-2008-003683 of the Ministry of Education of Spain. M. Moreto is funded by a MEC/Fulbright Fellowship.
Authors' addresses: C. Luque (corresponding author), Barcelona Supercomputing Center, Nexus II Building, Jordi Girona, 29, 08034 Barcelona, Spain; email: carlos.luque@bsc.es; M. Moreto, International Computer Science Institute, 1947 Center St., Berkeley, CA 94704; F. J. Cazorla, Barcelona Supercomputing Center, Nexus II Building Jordi Girona, 29, 08034 Barcelona, Spain; M. Valero, Computer Architecture Department, Universitat Politècnica de Catalunya, Jordi Girona 1-3, Office D6-201 Campus Nord, 08034 Barcelona, Spain.

DOI 10.1145/2400682.2400709 http://doi.acm.org/10.1145/2400682.2400709

1. INTRODUCTION

Thread-Level Parallelism (TLP) has been implemented in recent processors to overcome the limitations imposed when exploiting Instruction-Level Parallelism (ILP). TLP paradigms include a wide variety of processors. On one extreme of the spectrum, we find Simultaneous Multithreading (SMT) processors [Serrano et al. 1993; Tullsen et al. 1995], in which tasks share most of the processor resources. On the other end of the spectrum, Chip Multiprocessors [Olukotun et al. 1996] (CMP), in which tasks share only some levels of the cache hierarchy and the memory bandwidth. In between, there are other TLP paradigms like coarse-grain multithreading [Agarwal et al. 1991; Storino et al. 1998] or fine-grain multithreading (FGMT) [Halstead and Fujita 1988; Smith 1981][1]. Each of these designs offers different benefits as they exploit TLP in different ways, which motivates processor vendors to combine different TLP paradigms in their latest processors. Some notorious examples are the Intel core i7 [Rotem et al. 2011] and IBM POWER7 [Sinharoy et al. 2011], which are CMP+SMT processors, and the ORACLE UltraSPARC T4 [Oracle 2012] that is CMP+FGMT. This trend in integrating different TLP paradigms will keep growing in importance according to ITRS road map for the future years [ITRS 2011].

Combining several paradigms of TLP in a single chip allows improving system performance, but it also introduces complexities in the accounting of CPU utilization of running tasks. Per-task accounted CPU utilization affects several key components of a computing system [Luque et al. 2009; 2011; Eyerman and Eeckhout 2009]. For instance, if the OS scheduler is not able to properly account for the CPU utilization of each task, the OS scheduling algorithm will fail to maintain fairness between tasks. The OS can balance the time each task is scheduled onto a CPU, but not the CPU progress each task does when scheduled onto a CPU, since the amount of resources (i.e., cache space) a task receives is in general not under the control of the OS. As a consequence, the scheduling algorithm cannot guarantee that a task progresses according to its software-assigned priority. Also, accurate CPU time accounting can help in detecting good corunner tasks in a workload, improving system performance as a result. Finally, data centers charge customers according to the use of their resources. Having accurate per-task CPU utilization will facilitate server consolidation, allowing an efficient usage of the servers and a fair charging to users.

The main problem of CPU accounting in MT processors lies on the fact that the performance of a task depends on both the time the task runs and the amount of resources it receives during that time. The latter is in general not under the control of the user or the OS. To make things worse, there is a nonlinear relation between the percentage of resources assigned to a task and the slowdown it suffers with respect to running in isolation with all resources. To overcome this situation, hardware support has been proposed to improve the way in which CPU utilization is measured in pure-CMP [Luque et al. 2009, 2011] and pure-SMT processors [Eyerman and Eeckhout 2009; Gibbs et al. 2005]. In both cases, the focus is on providing a way to compute the slowdown that a task suffers when running on the MT processor due to the interaction with other tasks (*inter-task congestion* or *misses*).

In this article, we show that the combination of previous approaches either incurs an unaffordable hardware cost to track CPU utilization in processors with multiple TLP paradigms, or leads to inaccuracies in its measurement. Consequently, we introduce *Micro-Isolation-Based Time Accounting* (MIBTA), a new approach to compute CPU utilization in CMP+SMT processors. Instead of adding hardware support in *each* shared hardware resource to track tasks' slowdown, MIBTA makes use of a time sampling

[1]In this aticle the term multithreaded (MT) processor refers to any processor executing more than one thread simultaneously.

technique in which tasks run in isolation for short periods of time, with negligible effect on the system throughput and with high accuracy when measuring CPU time. In particular, our proposal combines the following two approaches.

—At SMT level, where tracking how threads interact in each core resource would introduce significant hardware overhead, our technique periodically runs each task in isolation to measure its CPU utilization. The execution of each workload is divided into two phases that are executed in alternate fashion. In the first, *isolation*, phase all tasks but one are stopped so that the IPC in isolation of the task is measured. In a second, *multithreaded*, phase all tasks run together. The ratio IPC_{MT}/IPC_{isol} times the total execution time gives the CPU utilization for each task. Since the number of available threads in each SMT processor is restricted (only 2–8 threads), this solution can be implemented with minimal performance degradation. Our experiments show that less than 1.1% and 1.8% throughput degradation is obtained for 2- and 4-way SMT processors, respectively.

—At CMP level, our technique makes use of dedicated hardware monitoring support to track the interferences between tasks running in different cores. This hardware support is based on an ITCA accounting mechanism [Luque et al. 2009; 2011], which tracks the conflicts in the last level of cache (LLC), shared among all different cores, and estimates with high accuracy the CPU utilization in CMP processors. In this article, we propose a new monitoring hardware that significantly reduces the storage overhead of ITCA without affecting its high accuracy. The Randomized Sampled Auxiliary tag directory, denoted RSA, combines sampling techniques with randomized algorithms to predict inter-task misses to the entire LLC. When integrating RSA with MIBTA, the inaccuracy is reduced from 8.9% to 5.3% in a single core processor.

MIBTA combines both proposals to provide tight CPU utilization accounting in CMP+SMT processors with small hardware overhead. Our results show that for an 8-core 2-way SMT configuration MIBTA leads to 5.0% inaccuracy when measuring CPU time, while previous approaches lead to inaccuracies between 21% and 79%. These results are consistent among all evaluated processor setups and a wide variety of workloads.

The rest of this article is structured as follows. Section 2 analyzes current CPU accounting approaches for pure-SMT and pure-CMP processors. Section 3 introduces our accounting approach for multi-TLP processors. Section 4 describes our experimental environment and Section 5 provides the experimental results. Section 6 discusses other considerations regarding CPU accounting, while Section 7 is devoted to the related work. Finally, Section 8 concludes this work.

2. BACKGROUND

2.1. Principle of Accounting

It has been shown that though the total execution (wall clock) time of a task running in a single-threaded uniprocessor system is mainly affected by the number of corunning tasks, the time accounted to that task (*sys+user* in Unix-like systems) is not affected by other tasks. That is, the time accounted to that task is always the same regardless of the workload in which it is executed, i.e., regardless of how many tasks are sharing the hardware resources at any given time [Luque et al. 2009]. This property is known as the *Principle of Accounting* and ensures that each task in a workload will be accounted a fair CPU time.

The main problem to provide the Principle of Accounting in MT processors is that the execution times of tasks are highly influenced by the on-chip shared resources, which tasks *compete for*. As a result, the execution time, and hence the CPU utilization, of a task does not only depend on the time it runs on the processor, as it is the case in uniprocessor systems, but also on the other tasks it runs with (the workload). The workload in which a task runs determines the inter-task conflicts accessing shared resources it suffers. In general, the OS has no direct control on how resources are distributed among tasks, and hence the interaction they suffer. Moreover, the nonlinear relation between the resources a task receives and the progress it does makes the computation of CPU utilization hard [Luque et al. 2009; Eyerman and Eeckhout 2009; Fedorova et al. 2007].

The main particular problem to solve by CPU account mechanisms can be formulated as follows: A CPU accounting mechanism has to determine dynamically, while a task X is simultaneously running with other tasks, the time it would take X to execute the same instructions if it was alone in the system. If this can be accurately determined, then the CPU utilization of each task can be computed at context switch boundary. For instance, let's assume that a task X runs for a period of time in an MT processor, TR^{MT}_{X,I_X}[2], in which it executes I_X instructions. The actual time to account this task TA^{MT}_{X,I_X}, is the time it would take this task to execute in isolation these I_X instructions in the same architecture, denoted TR^{isol}_{X,I_X}. This would make the CPU accounting of a task independent from the rest of the workload, regaining the Principle of Accounting for MT processors.

We say that an accounting mechanism leads to *overestimation* (or *overaccounting*), when it accounts a task more time than the time it takes them to make the same progress when run in isolation $TA^{MT}_{X,I_X} > TR^{isol}_{X,I_X}$, and *underestimation* (or *underaccounting*) in the opposite situation. Both under- and overestimation are inaccurate measurements of CPU utilization. We use the term *off estimation* to define such inaccuracy when measuring CPU utilization.

2.2. Current Accounting Methodologies

The Classical Approach (CA) for CPU accounting is inherited from uniprocessor systems, where the OS does not consider the interaction between tasks caused by hardware shared resources. With the CA, the time accounted to task X in an MT processor, TA^{CA}_{X,I_X}, can be expressed as $TA^{CA}_{X,I_X} = TR^{MT}_{X,I_X}$. The CA accounts tasks based on the time they run on a CPU, and not by their resource utilization. Here, the implicit assumption is that all tasks have full access to the processor resources when running. However, tasks share many hardware resources with other tasks in an MT processor. Ergo, tasks take longer to finish executing and are overaccounted with the CA.

The IBM POWER^{TM} processor family includes multicore processors in which each core is SMT. POWER5/6/7 include a per-task accounting mechanism called Processor Utilization Resource Register (PURR) [Broyles et al. 2011]. The PURR approach estimates the CPU time of a task based on the number of cycles the task can decode instructions: each 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, all tasks running on the same core are accounted $1/N$ of the cycle, where N is the number of running tasks in the SMT. PURR represents a good solution when running tasks are ILP bound, but has more problems when the workload contains memory-bound tasks.

Figure 1 shows the accuracy of the CA and the PURR when measuring the CPU time accounted to running tasks in 2- and 4-task workloads on a POWER5 processor (the lower the better). We show the results for workloads that run on a single-core 2-way

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

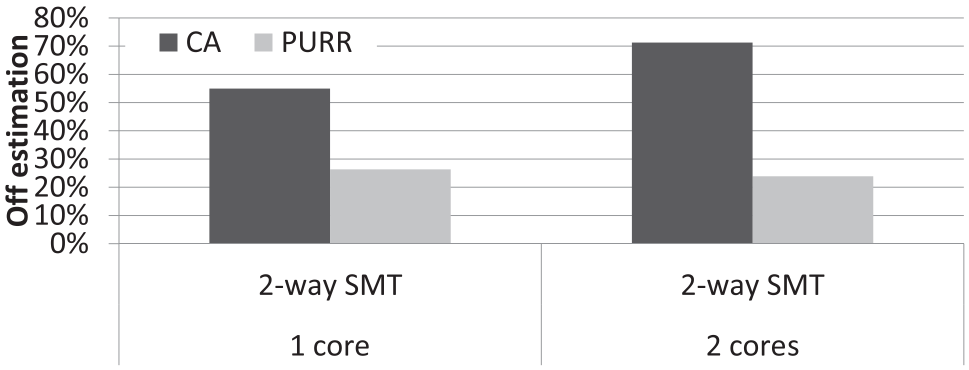

Fig. 1. Measured accounting accuracy for the CA and PURR on a POWER5 processor.

SMT and on a 2-core 2-way SMT configuration. We observe that PURR significantly improves the accuracy provided by the CA, reducing its inaccuracy from 55% to 26% in the single-core configuration. In the 2-core configuration, the CA shows even higher inaccuracies (reaching 71%), while PURR results are more stable (close to 24%). Even if the results of PURR are more accurate than the CA, there is still room for improvement to narrow down CPU time accounting inaccuracies.

2.3. Solutions for CMP Processors

Several authors have shown that the main source of inaccuracy when measuring CPU time in single-threaded CMP processors are inter-task misses in the LLC on chip [Luque et al. 2009; 2011]. For instance, when tasks A and B execute in different cores in a CMP processor, it may happen that task B evicts some data belonging to A from a shared level of cache. As a consequence, some accesses of task A that would hit in this cache level become inter-task misses in CMP mode. These inter-task misses cause task A to stall its execution until they are resolved, increasing the number of execution cycles required to process the memory access.

Luque et al. [2009, 2011] address this problem by adding hardware monitoring support to track inter-task misses in the LLC of single-threaded CMP processors. This CPU accounting mechanism stops accounting cycles to a task when it suffers an inter-task LLC miss and its pipeline is stalled. With this simple solution, the off estimation on an 8-core processor is reduced from 16% with the CA to 2.8% [Luque et al. 2011]. The storage cost of ITCA for the highest accuracy is not negligible, since a full copy of the tags of the LLC is required per task. In their original configuration, this structure required 30KB storage overhead. Reducing the stored bits of the tags and sampling techniques were proposed to reduce this overhead to 8KB at the cost of less accurate CPU time measurements.

2.4. Solutions for SMT Processors

In SMT processors, many more hardware resources are shared among tasks and, consequently, inter-task contention affects the performance of running tasks. Eyerman and Eeckhout [2009] propose a new cycle accounting architecture for SMT processors based on estimating the CPI stack of each running task [Eyerman et al. 2006]. This proposal fully controls all the components that affect the speed of a task. To that end, the solution in Eyerman and Eeckhout [2009] tracks fifteen different components of the CPI stack with dedicated hardware. The CPI components related to the cache hierarchy are scaled based on the conflicts the task suffers. To that end, authors use significant storage per task and more importantly, dedicated logic, to compute the scaling of each

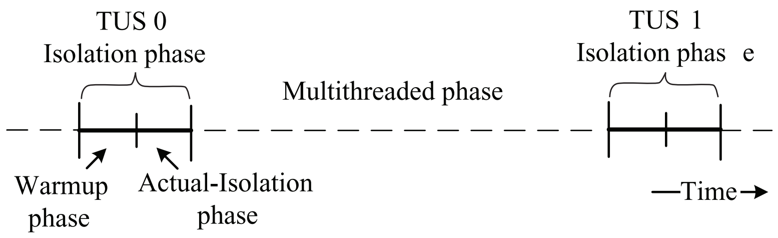

Fig. 2. The isolation and multithreaded phases in MIBTA mechanism.

CPI component on a per-cycle basis. Overall, this solution provides detailed information of the execution of each task at the cost of more complex structures (to track all possible events), logic, and dedicated floating-point ALUs. The main assumption in this approach is that the ReOrder Buffer (ROB) is the main bottleneck, in which case the solution provides tight cycle accounting: average 7.2% and 11.7% off estimation in a 2- and 4-way SMT processor, respectively [Eyerman and Eeckhout 2009]. However, other core resources such as the issue queues or the register files are also a usual bottleneck for performance. In that case, the accuracy of this solution significantly degrades, as we show in Section 5.6.

3. CPU ACCOUNTING IN MT ARCHITECTURES

As previous proposals do, we advocate for using the *execution progress* (or *slowdown*) that a task does in the MT processor as a proxy to determine its execution time in isolation, and hence the CPU time to account to it. Let's assume that a task X runs for a period of time in an MT processor, TR^{MT}_{X,I_X}, in which it executes I_X instructions. The relative progress that task X has in this interval of time can be expressed as $P^{MT}_{X,I_X} = TR^{isol}_{X,I_X}/TR^{MT}_{X,I_X}$ (the slowdown is the inverse of the progress). The relative progress can also be expressed as $P^{MT}_{X,I_X} = IPC^{MT}_{X,I_X}/IPC^{isol}_{X,I_X}$ in which IPC^{MT}_{X,I_X} and IPC^{isol}_{X,I_X} are the IPC of task X when executing the same I_X instructions in the MT processor and running alone, respectively. Then, $TA^{MT}_{X,I_X} = TR^{isol}_{X,I_X} = TR^{MT}_{X,I_X} \cdot P^{MT}_{X,I_X}$ that fulfills the Principle of Accounting.

3.1. Micro-Isolation-Based Time Accounting for SMT Processors

Tracking each resource utilization in an SMT processor introduces high hardware cost, complicates its design, and is architecture dependent. Moreover, accounting mechanisms do not directly contribute to improve system performance, which motivates the use of an accounting mechanism as simple as possible. In order to estimate the CPU utilization in SMT, we introduce a *Micro-Isolation-Based Time Accounting* (MIBTA) mechanism. Unlike previous proposals, MIBTA does not track task interaction in each shared resource in an SMT processor. MIBTA divides the execution of running tasks into two phases that are executed in alternate fashion, as shown in Figure 2.

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

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

Even if the TUS is run in isolation, it may still suffer inter-task conflicts in shared resources as in the precedent *MT* phase all tasks used those shared resources. In our simulated architecture, as described in Section 4, there are the following shared core resources[3]: fetch and issue slots, issue queue entries, physical registers, caches, TLBs, and the branch predictor. Private per-core first-level instruction and data caches, TLBs, and the branch predictor can suffer destructive interference because an entry given to a task can be evicted by another task before it is accessed again. In order to get more insight into this interference, we have measured the average inter-task conflicts the (Task Under Study) TUS suffers during the *isol* phase. We have observed that, as the *isol* phase progresses, the TUS evicts all data from other tasks. Consequently, the number of conflicts goes toward zero for the instruction cache, data cache, TLBs, and the Branch Target Buffer (BTB). We observed that 50,000 cycles after the beginning of the *isol* phase, most interference in these shared resources is removed. The branch predictor, Pattern History Table (PHT),takes much longer to clear: We have measured that it takes more than 5 million cycles before misses due to the interaction with other tasks (inter-task misses) have disappeared. However, this interference is mostly neutral, giving a negligible loss in the branch predictor hit rate of less than 1%. Hence, we ignore the interference in the branch predictor.

To remove inter-task conflicts in all core shared resources, we propose to split each *isol* sample into two subphases. During the first subphase, the *warmup phase*, that consists of 50,000 cycles, the TUS is given all resources, but its IPC is not measured. In the second subphase, the *actual isolation phase*, the TUS keeps all resources, and its IPC is measured. The duration of this subphase is 50,000 cycles. In Section 5, we study the accuracy of MIBTA with different warmup and actual isolation phase lengths.

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

Hardware implementation. The implementation of MIBTA requires reduced hardware support. In fact, many current processors already incorporate similar support that could be used to provide MIBTA's required functionality. MIBTA needs a countdown timer that is programmed to trigger at the end of each phase: warmup, actual isolation, and MT phases. The 3 registers that save these values, *wuReg*, *aiReg*, and *mtReg*, can be made visible and writable from the OS.

MIBTA also requires that only one task runs during each *isol* phase. This can be done in a straightforward way by stopping the fetch of instructions of the other tasks in the core. Processors such as the IBM POWER7 already incorporate, hardware thread priority mechanism [Sinharoy et al. 2011]: When a thread is assigned the lowest priority it is allowed to fetch instructions only once every dozen of cycles. MIBTA would simply require another priority level in which a thread is not allowed to fetch further instructions until its priority is changed back to its previous value.

After stopping the fetch of instructions for a given task, its in-flight instructions will eventually commit before the end of the warmup phase. The only information that task would have in the core is its program counter and the architectural state registers. In processor architectures where the architectural registers are kept in a different register file than the physical registers, MIBTA can be implemented just with the small change in the fetch stage mentioned above. In processors in which the architectural and physical registers share the same register file (the most common case), it is necessary to

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

deallocate from the register file the architectural registers of the nonrunning tasks in the isolation phase so that the TUS enjoys as many resources as when actually running in isolation. We do this, with small changes in the pipeline. In particular, we propose a mechanism that releases architectural registers of nonrunning tasks and locks them into the LLC cache. We denote this solution *Register File Release* (RFR) mechanism. A similar approach is implemented in the Intel Sandy Bridge processor [Rotem et al. 2011], where the state of the machine in a given core can be flushed to the LLC cache to turn off the core and reduce system energy. This operation takes in the order of hundreds of cycles, and deploys the data path already implemented in the processor, not requiring extra wires for data. Overall, MIBTA works as follows.

—Just after the MT phase ends (and the warmup phase starts), we stop fetching instructions from all the tasks except the TUS. When there are no instructions from the other tasks in the ROB, we store the information from the architectural registers of the other tasks in the LLC cache, locking the corresponding cache lines. The time at which the other tasks have no in-flight instructions can be determined by deploying performance counters present in many current architectures that are able to measure it. Afterwards, the architectural registers from other tasks are released, increasing the number of renaming registers available to the TUS.
—At the end of the warmup phase, the number of instructions executed is saved into the register *wuInstr*.
—When the actual isolation phase ends, the instructions executed are saved into another register, *aiInstr*. The number of instructions executed during the actual isolation phase is $I_{isol,X} = aiInstr - wuInstr$. All architectural registers are loaded from the LLC cache back into the register file, and the normal MT phase begins.

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

3.2. Inter-Core Conflict-Aware Accounting for CMP+SMT Processors

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

To overcome this problem, MIBTA has to detect every time the TUS suffers an inter-task LLC miss and take into account this information in the final accounting of the task. Previous proposals make use of an *Auxiliary Tag Directory* (ATD) per task to track inter-task misses [Qureshi and Patt 2006; Luque et al. 2009]. The ATD has the same associativity and size as the tag directory of the shared LLC and uses the same replacement policy. It stores the behavior of memory accesses per task in isolation. While the tag directory of the LLC 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 LLC cache and hits in its ATD, we know that this memory access would have hit in cache if the task had run in isolation [Mattson et al. 1970]. Thus, it is identified as an inter-task LLC miss. Otherwise, it is identified as an intra-task miss intrinsic to the task.

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

A second option consists on considering a sampled version of the ATD, denoted sATD [Luque et al. 2009; Qureshi and Patt 2006], which only monitors some sets of the LLC cache and obtains the miss rate in isolation. Under this approach we track inter-task misses to the sampled sets. However, when the number of sampled sets is reduced, accuracy significantly decreases since we are not detecting inter-task misses to nonmonitored sets.

Based on the fact that sampled ATDs are very accurate in predicting LLC miss rates [Qureshi and Patt 2006], we propose to track the probability of having an inter-task miss in the sampled sets, i.e., the ratio between inter-task misses and total misses to the sATD of the task. During the *MT* phase, the number of inter-task and total misses per task are tracked and accumulated in two registers per task. When the *isol* phase begins, the ratio between these values is computed and stored as a 10-bit integer multiple of $\frac{1}{1024}$ (0 represents 0, 512 represents 0.5, 1024 represents 0.999, and so on and so forth). This threshold is always computed during warmup phase. During actual isolation phase, the same inter-task miss probability is assumed for accesses to nonmonitored sets. When missing on a non-monitored set, a 10-bit random number is generated with a Linear Feedback Shift Register (LFSR) [Beker and Piper 1982]. If this number is less than the previously obtained threshold, this LLC miss is predicted to be an inter-task miss. Otherwise, we assume it is an intra-task miss. This Randomized version of the Sampled ATD, denoted `RSA`, predicts inter-task misses to these nonsampled sets. In Section 4, we present a detailed study of the accuracy of all these possible implementations, concluding that `RSA` provides nearly the same accuracy as the entire ATD with the same hardware cost as a sampled ATD.

We add a bit in each entry of the Miss Status Holding Register (MSHR) to track inter-task misses. This bit is set to one when we detect an inter-task data miss with one of the described tracking logic. 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 in parallel the LLC tag directory and the inter-task miss detection logic of the task. If we have a miss in the LLC tag directory and a hit in the tracking logic, we know that this is an inter-task LLC cache miss and we set the bit in the MSHR entry to 1. Once the memory access is resolved, we free its entry in the MSHR.

Accounting CPU time. To determine whether the task is progressing in a given cycle of an actual isolation phase, we make use of the decision provided by the ITCA mechanism [Luque et al. 2009, 2011]. This mechanism is specifically developed for accounting in CMP architectures with shared caches. ITCA stops accounting to a task in two situations: (1) when the ROB is empty because of an inter-task LLC cache instruction miss, and (2) when the instruction in the top of the ROB is an inter-task LLC miss and the register renaming stage is stalled. In other situations without inter-task LLC misses or when inter-task misses overlap with intra-task misses, the accounting is not stopped since the task is performing similar progress it would make in isolation.

Figure 3 shows a sketch of the hardware implementation of MIBTA that is the same as for ITCA [Luque et al. 2011]. Only four *Hardware Resource Status Indicators* (HRSI)

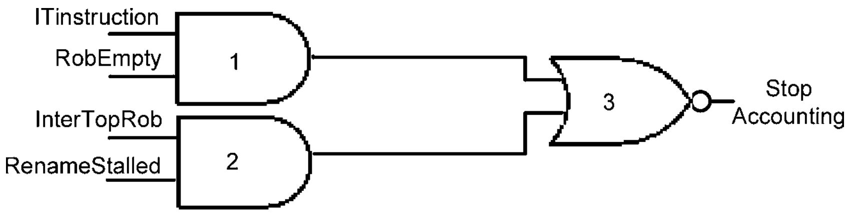

Fig. 3. Logic to stop accounting required for MIBTA

are required to decide whether a cycle should be accounted to a task or not. The HRSI *ITinstruction* indicates whether the task has an inter-task LLC cache instruction miss or not, while *RobEmpty* indicates if the ROB is empty or not. The output of gate 1 indicates if the ROB is empty due to an instruction inter-task LLC cache miss. The HRSI denoted *InterTopRob* tracks if the oldest instruction in the ROB suffered an inter-task LLC cache data miss, while *RenameStalled* monitors if the register renaming is stalled. The output of gate 2 indicates if the machine is stalled due to an inter-task LLC cache miss. Finally, if any of the gates (1) or (2) returns 1, we stop the accounting. Otherwise, we account the cycle normally to the task since it is progressing as in isolation.

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

These registers are read-only like the *timestamp register* in Intel architectures, and can be communicated to the OS. On every context switch, the OS reads for each task X the ICR_X, IIR_X, $MTIR_X$ and $MTCR_X$ registers. With this information, the OS estimates the time to account to each task as: $TA_{X,I_X} = ICR_X + \frac{IPC_{MT,X}}{IPC_{isol,X}} \cdot MTCR_X$, where $I_X = IIR_X + MTIR_X$ is the total number of executed instructions, the IPC in isolation $IPC_{isol,X} = \frac{IIR_X}{ICR_X}$, and the IPC as part of the workload $IPC_{MT,X} = \frac{MTIR_X}{MTCR_X}$. At the context-switch boundary, in fact on every clock tick, the OS also updates metrics of the system and carries out the scheduling tasks. Thus, the OS could potentially use the information provided by MIBTA to find better coschedulers, similarly to Fedorova et al. [2007]. When a task is swapped out, its associated ICR_X, IIR_X, $MTIR_X$, $MTCR_X$ can be updated in the *task struct* and are reset before the next task starts.

4. EXPERIMENTAL ENVIRONMENT

Processor architecture. We use MPSim [Acosta et al. 2009], a CMP+SMT simulator derived from SMTsim [Tullsen et al. 1995]. The baseline configuration is shown in Table I, which represents an out-of-order processor with an 11-stage-deep pipeline. In our baseline architecture, up to 8 instructions from a single task are fetched from the instruction cache in program order. We use *icount* fetch policy to determine from which of the available tasks instructions are fetched. Next, instructions are decoded and renamed in order to track data dependences. After renaming, an entry in the Issue Queues (IQs) is allocated to each instruction until all operands are ready. Each instruction also allocates an ROB entry, and a physical register, if required. ROB entries are assigned in program order. When an instruction has all its operands ready, it is issued[4]: it reads its operands, executes, writes its results, and finally commits. Data

[4]In this work, the term *issue* is applied to the action of submitting instructions from the issue queues to the back-end of the machine.

Table I. Simulation Configuration

Core configuration		
	2-way SMT	4-way SMT
Number of core	1,2,4,8	1,2,4
Issue Queue entries	48 int, 48 fp, 48 ld/st	64 int, 64 fp, 64 ld/st
Physical Registers	164 int, 164 fp	256 int, 256 fp
ROB size	256	352
Execution Units	4 int, 2 fp, 2 ld/st	
Fetch Policy	ICOUNT 1.8	
Branch predictor	2K entries, gshare	
Branch Target Buffer	256 entries and 4 ways	
Clock Frequency	2.0GHz	

Cache/Memory Configuration				
Core/s	1	2	4	8
LLC (shared)	2MB	4MB	8MB	16MB
	16 ways, 8 banks, 128 Bytes			
Instruction (per core)	64 KB, 4 ways, 1 bank, 128 Bytes			
Data (per core)	64 KB, 8 ways, 1 bank, 128 Bytes			
ITLB (per core)	128 entries, 8 KB page			
DTLB (per core)	256 entries, 8 KB page			
Latencies	LLC (15), Memory (300)			

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

The main shared resources in our baseline core architecture are the following: (1) the front-end bandwidth, which is assigned to tasks according to the instruction fetch policy; (2) the IQs are shared between all tasks running on a core; (3) the issue bandwidth: we use is *first in first out* (oldest first) as issue policy; (4) the physical register file is common to all tasks; (5) instruction and data caches are shared between tasks, although the data of one task is not shared with any other task; (6) instruction and data TLB are also shared and tagged with task identifier; and (7) the ROB is shared among all tasks in a core. At the chip level, the main shared resources among all tasks are: (1) LLC unified cache; (2) the memory controller: we use First Come First Served policy; and (3) memory bandwidth.

Workloads selection. We use several processor setups: four core counts (1, 2, 4, and 8), and 2- and 4-way SMT cores, for a total of 7 different configurations[5]. We feed our simulator with traces collected from the whole SPEC CPU 2006 benchmark suites using the reference input set. Each trace contains 100 million instructions, selected using SimPoint methodology [Sherwood et al. 2001]. From these benchmarks, we generate different workloads. In each workload, the first task in the tuple is the Principal Thread (PTh) and the remaining tasks are considered Secondary Threads (SThs). In every workload, we execute the PTh until completion. The other tasks are reexecuted until PTh completes. Running all N-task combinations is infeasible as the number of combinations is too high. Hence, we randomly generate 26 workloads for each evaluated configuration.

Performance metrics. As the main metric, we measure how off is the estimation done by an accounting approach for the PTh ($TA_{PTh,I_{PTh}}$) from the actual time it should be accounted ($TR^{isol}_{PTh,I_{PTh}}$). We call *off estimation* the ratio $\left|1 - (TA^{MT}_{PTh,I_{PTh}}/TR^{isol}_{PTh,I_{PTh}})\right|$.

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

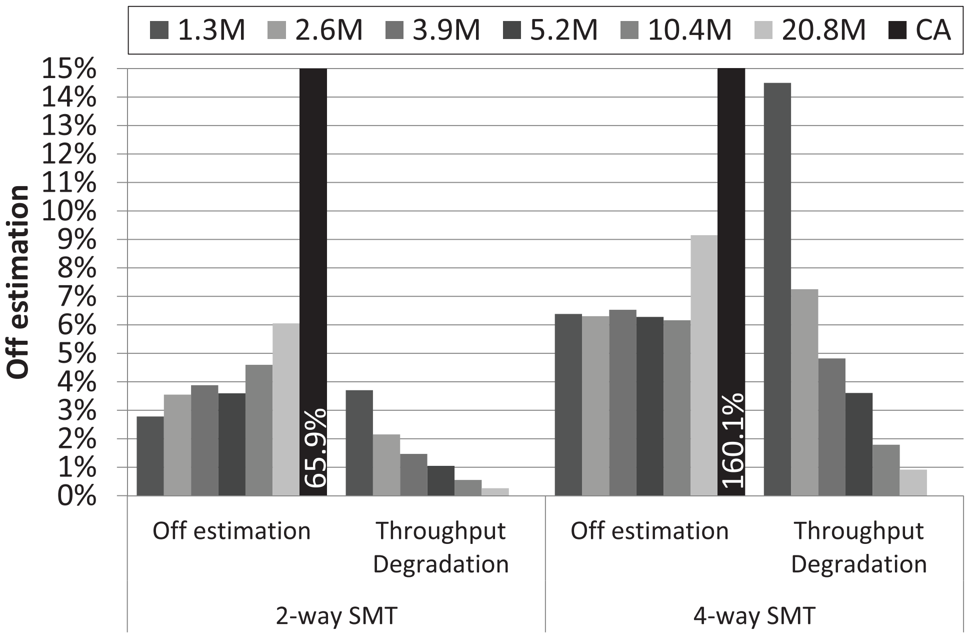

Fig. 4. MIBTA off estimation and throughput degradation on an SMT processor under different sampling intervals.

For each workload (N tasks) we also measure its throughput, which is the sum of the progress of each task (weighted speedup): Throughput $= \sum_{i=0}^{N-1} P_i^{MT} = \sum_{i=0}^{N-1} \frac{IPC_i^{MT}}{IPC_i^{isol}}$.

5. EXPERIMENTAL RESULTS

We perform several studies to evaluate the accuracy of MIBTA. First, we focus on a single-core processor to determine the best design parameters for the MIBTA accounting mechanism. Afterwards, we evaluate different implementations of our proposal that minimize the storage overhead of MIBTA without decreasing its accuracy. Finally, we evaluate MIBTA in a CMP+SMT configuration with two, four, and eight cores, and compare its results with previously proposed accounting mechanisms.

5.1. Sensitivity Analysis for Single-Core Architectures

We determine the design parameters that provide the best trade-off in terms of accuracy and throughput in a single-core processor. When moving to a CMP+SMT scenario, similar results will be obtained, as we show in Section 5.4.

Figure 4 shows the off estimation and throughput degradation results for a 2- and 4-way SMT processor under different sampling intervals, ranging from 1.3 to 20.8 million cycles. The sampling interval is the number of cycles between two *isol* phases of a given task. We use sampling intervals from 1.3 to 20.8 million cycles. In both configurations, the off estimation increases with the sampling interval, since MIBTA cannot capture some phase changes of the task. In contrast, the throughput degradation decreases with the sampling interval, since there are less *isol* phases that degrade total throughput. With all sampling intervals, the off estimation of MIBTA clearly improves over CA. With the lower sampling intervals, off estimation reaches just 2.8% and 6.4% in 2- and 4-way SMT processors, respectively.

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

We also explore off estimation for different warmup and actual isolation phases lengths. For a constant length of the isolation phase from 50 to 100 thousand cycles, the best results are obtained with balanced warmup and actual isolation phases. In contrast, the worst results are obtained with extreme values, when either the warmup or the actual isolation phase is only 10 thousand cycles. On average for 2- and 4-way SMT processors, the optimal results are obtained with warmup and actual isolation phases of length 50 thousand cycles with 4.8% off estimation. Several configurations are close to this optimal value, with the worst results (6.2% off estimation) obtained with warmup and actual isolation phases of 80 and 10 thousand cycles, respectively. In the remaining experiments, we maintain 50 thousand cycles for both warmup and actual isolation phases.

Next, we explore which resource conflicts lead to the off estimation obtained with MIBTA on the SMT processor. When comparing the execution of each benchmark in isolation and in the actual isolation phase, we detect that LLC conflicts are the key contributor to the off estimation of MIBTA. Even in a scenario with a perfect LLC cache (without any inter-task LLC cache conflicts), the off estimation would be only reduced by an extra 1% and 1.2% in the 2-way and 4-way SMT configurations. The remaining off estimation is explained by the sampling interval, since MIBTA can not capture all task phases. Finally, we discover that in the 4-way SMT configuration, the register file suffers a significant increase in inter-task conflicts: In the isolation phase, the architectural state of all tasks is still stored in these registers, and the task under study can not make use of these resources. This finding motivates the use of the register release mechanism that we evaluate in Section 5.3.

5.2. MIBTA Storage Overhead

Next, we evaluate different implementations of MIBTA with different storage overheads devoted to tracking inter-task LLC misses. Figure 5 shows the off estimation results of the four possible implementations in 2- and 4-way SMT processors. Without using any storage (`no ATD` configuration in Figure 5), MIBTA reduces off estimation from 113% to just 9.0% on average. The warmup phase effectively eliminates the majority of inter-task conflicts and, even if some inter-task LLC misses are not detected, the off estimation is heavily reduced.

When considering 32 sampled sets out of the 1024 sets of the LLC cache (`sATD` configuration), the storage overhead supposes just 960 bytes per task. However, very few inter-task misses are detected and, consequently, off estimation is very close to the configuration without ATDs. In contrast, when tracking the inter-task ratio with the randomized sampled ATD (`RSA` configuration), the off estimation is reduced to 5.3% on average, very close to the 4.9% average off estimation with the entire ATD. Implementing `RSA` requires a sampled ATD with 32 sampled sets, two 64-bit registers, two shifter registers, an LFSR, and four 64-bit special-purpose registers. In our current configuration, the total storage overhead per task is 1KB. At core level, one bit per entry in the ROB and the MSHR are required, as well as three 20-bit registers. This supposes an extra 0.04KB and 0.05KB in 2- and 4-way SMT cores, respectively.

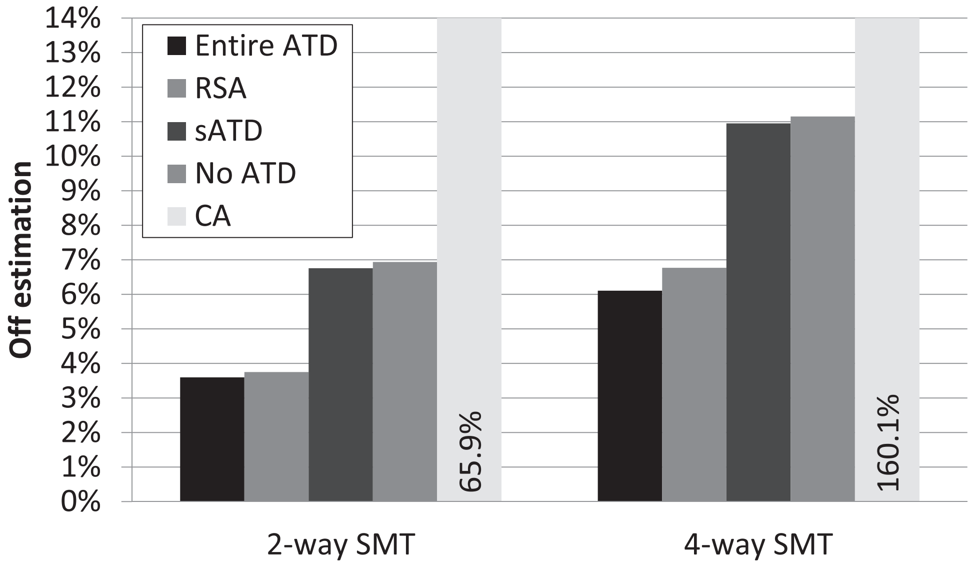

Fig. 5. MIBTA off estimation in 2- and 4-way SMT processors using different storage overheads.

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

5.3. Shared Register File

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

Figure 6 shows the results in a 2- and 4-way SMT processor setup when using the original MIBTA proposal with and without the register release mechanism. This mechanism is combined with the randomized version of the sampled ATD. The average error in these configurations is reduced to 3.2% and 5.2%, respectively. The off estimation reduction is significant in the 4-way SMT configuration, since the contention in the shared register file is much higher than in the 2-way SMT configuration (100 and 160 available register out of 164 and 256, respectively). In fact, the accuracy in that configuration is 23.5% better than with the entire ATD but without the RF release mechanism (5.2% instead of 6.8%). In all configuration, the extra throughput degradation due to this mechanism is insignificant (less than 0.05%).

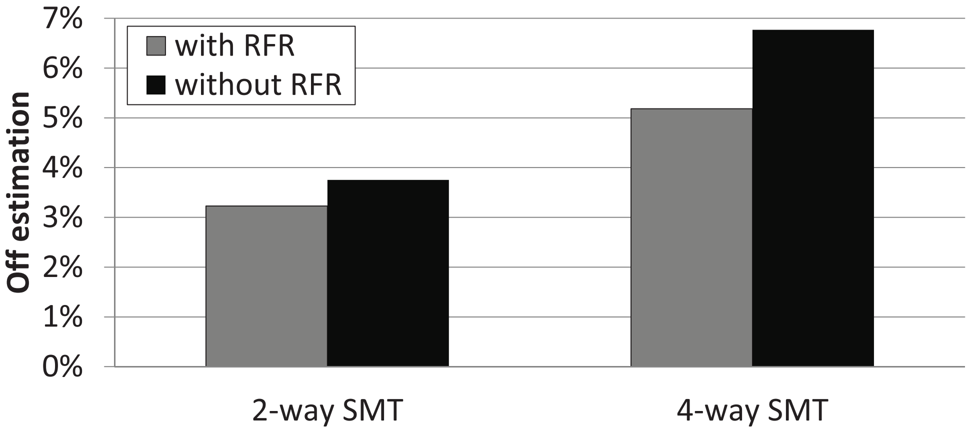

Fig. 6. Accuracy with/without the register file release (RFR) mechanism.

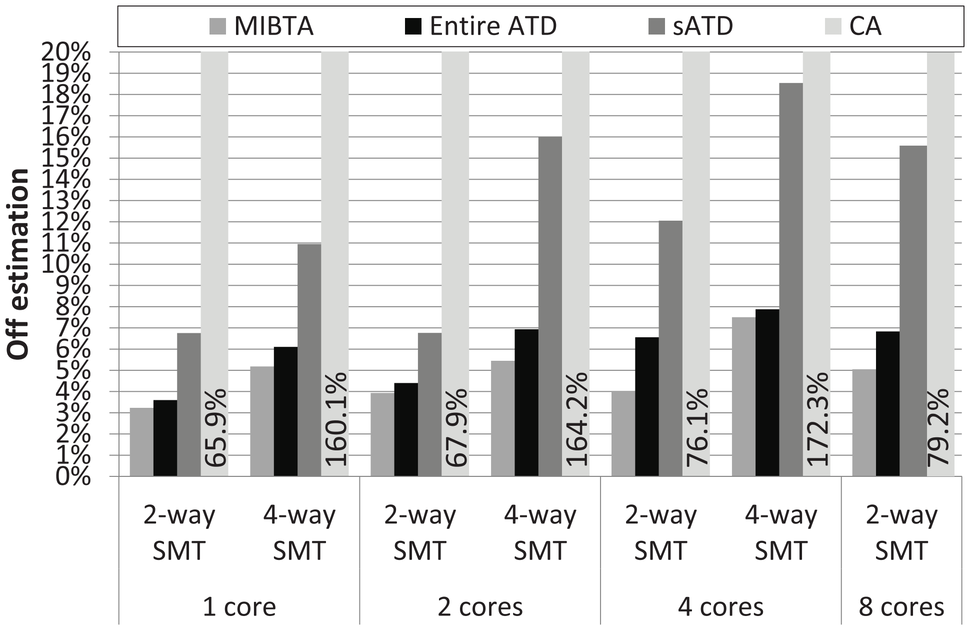

Fig. 7. MIBTA off estimation for 7 different CMP+SMT configurations.

5.4. MIBTA on CMP+SMT Architectures

Next, we move to a CMP+SMT scenario, with up to 8 cores sharing the LLC cache and memory hierarchy. In this case, during each isolation phase only one task is running on each core, removing on-core inter-task conflicts after the warmup phase. However, there will be still some inter-task conflicts when accessing the LLC and the memory hierarchy. Figure 7 shows the off estimation results for the seven different configurations. MIBTA has an off estimation under 5.0% in all 2-way SMT processors, while for 4-way SMT processors, the off estimation is always between 5.2% and 7.5%. MIBTA obtains better results than using the entire ATD due to the RFR mechanism and with a much lower hardware overhead. When using a sampled ATD, a solution with similar hardware overhead, the off estimation quickly raises to 18.5% and 15.6% in 4-core 4-way SMT and

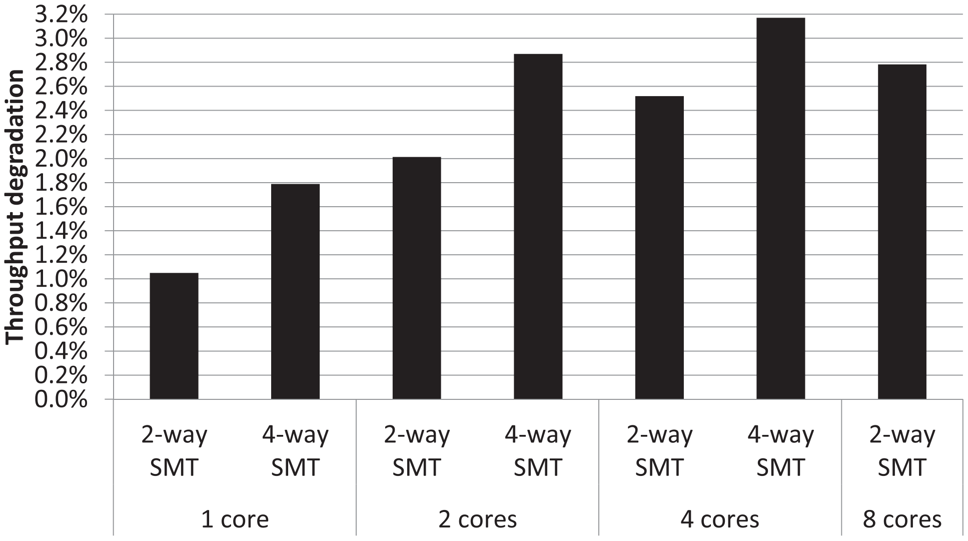

Fig. 8. MIBTA system performance degradation for 7 different CMP+SMT configurations.

8-core 2-way SMT configurations, respectively. In contrast, MIBTA has a more stable accuracy, suggesting that the memory contention is correctly addressed by MIBTA.

Note that MIBTA takes into account memory bandwidth conflicts when inter-task conflicts are in-flight. Bus, bank, and memory conflicts are not considered by MIBTA when there are only intra-task conflicts. However, our results confirm that the effect of those conflicts is small on CPU accounting accuracy, making it not worthy to devote extra hardware to track them. We elaborate more on this point on Section 5.5.

As mentioned before, the throughput degradation of MIBTA basically depends on the number of tasks that are stopped in each core during isolation phases. Figure 8 shows the obtained results in a CMP+SMT scenario. All values are below 3.2% and do not significantly increase with the number of cores in the system.

5.5. Memory Bandwidth Sensitivity

As mentioned earlier in this section, the memory bandwidth has not been identified as a main source of interaction between tasks in our different processor setups. To illustrate this point, we measure the memory bandwidth requirements of the evaluated workloads in all processor configurations. Figure 9 shows the percentage of workloads that require a given memory bandwidth to reach their maximum performance. Note that in all our processor setups, we keep the same ratio of LLC cache per core: 2MB/core.

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

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

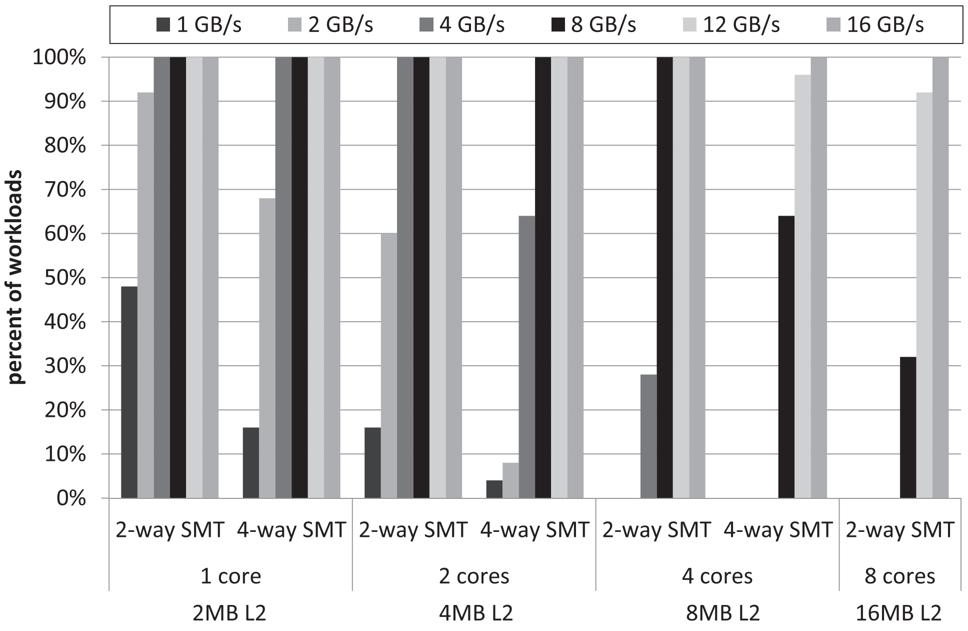

Fig. 9. Memory bandwidth requirements for 7 different CMP+SMT configurations.

5.6. Comparison with Other Accounting Mechanisms

Next, we compare the accuracy of MIBTA with previously proposed accounting mechanisms. These mechanisms, described in Section 2, are also implemented in our simulator. Figure 10 evaluates the accuracy of the CA, ITCA, PURR, Eyerman's proposal, and MIBTA across multiple processor configurations. The CA shows the worst results 112% average off estimation, since it is not aware of any inter-task conflict. ITCA also has similar results since it was designed for pure-CMP systems and does not take into account inter-task core conflicts, with an average off estimation of 102%.

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

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

In contrast, MIBTA shows much more accurate and consistent results across all configurations, with average errors between 3.2% and 7.5%. The results shown in

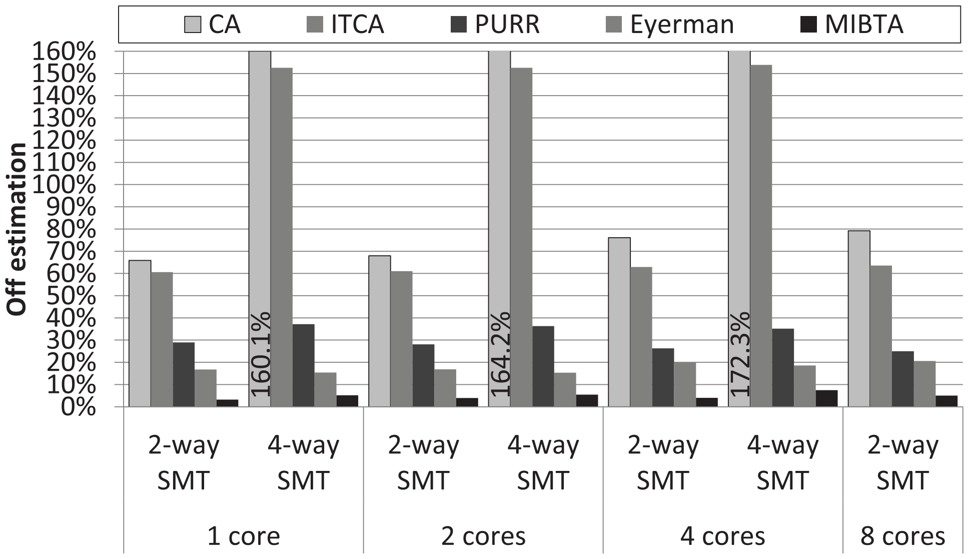

Fig. 10. Off estimation with different accounting mechanisms across several processor configurations.

Figure 10 indicate that it is important to consider other shared resources such as the issue queues or the register files. Thus, an approach such as MIBTA leads to more accurate results independently of the processor configuration in which it is run.

Even if previous proposals do not suffer any performance degradation (or negligible), the experiments in Section 5.4 show that the performance degradation introduced by MIBTA is very low: less than 3.2% in all configurations. In terms of implementation cost, the CA and PURR require negligible hardware support. ITCA and MIBTA require slightly more hardware support (basically the sampled ATD and the RSA, respectively), while Eyerman's proposal presents a nonnegligible hardware cost and complexity as the required hardware blocks are spread throughout all the pipeline of the processor. In contrast, MIBTA relies on isolation phases to predict performance in isolation without introducing expensive hardware support.

6. OTHER CONSIDERATIONS

6.1. System-Level Considerations

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

To exemplify the potential benefits of MIBTA for the software stack, we perform the following experiment: We combine the feedback provided by MIBTA with the symbiotic scheduling techniques introduced by Snavely and Tullsen [2000] and Snavely et al. [2002]. The SOS scheduler (for Sample, Optimize, Symbios) combines a sample phase which collects information about various possible schedules, and a symbiosis phase which uses that information to predict which schedule will provide the best performance. The sampling phase of the SOS scheduler is very different from the isolation phases of MIBTA: SOS samples some random schedules from the huge amount of possible schedules in a workload. After the sampling phase, SOS predicts the best candidate

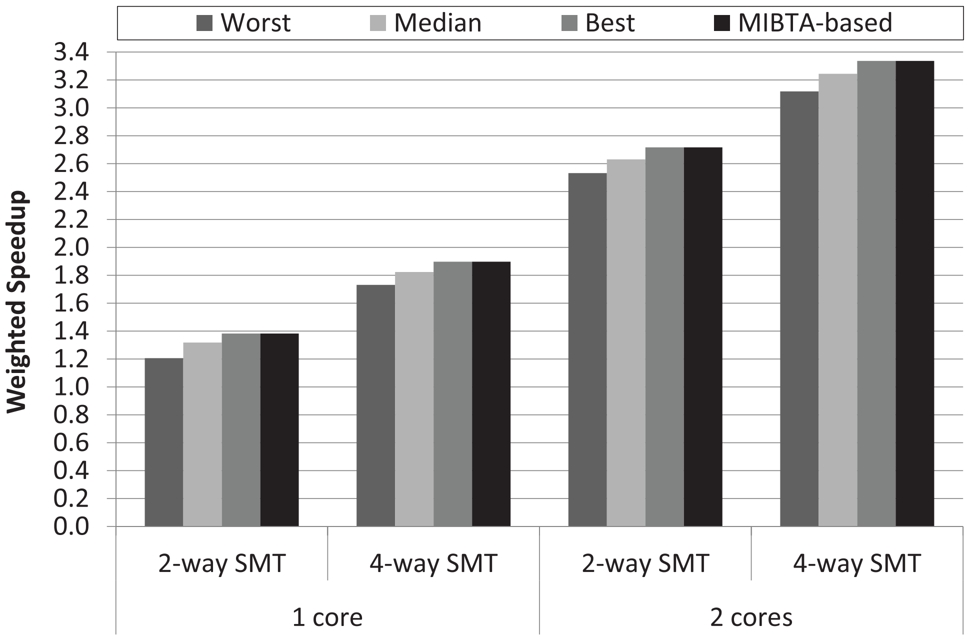

Fig. 11. Weighted speedup of different symbiotic schedulers for four CMP+SMT configurations.

schedule based on different performance counters measurements. Finally, it runs the candidate optimal schedule for the remaining time.

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

Figure 11 shows the weighted speedup results with different schedules in four processor configurations with different number of cores (1 and 2), and 2- and 4-way SMT cores. In each configuration, we randomly select a workload with more tasks than available hardware threads. The number of total tasks in the workload is 16 for the 2-core 4-way SMT configuration, and 8 for the remaining ones. SOS generates 20 random schedules (this number is significantly smaller than the total number of available schedules) and decides the candidate optimal schedule for the remaining time. We evaluate the heuristic based on the information provided by MIBTA, and we also report the performance of the worst, median, and best schedule out of the 20 random schedules. We can see that the range of values of the weighted speedup is wide: the best schedule obtains between 7.0% and 14.6% higher weighted speedup than the worst schedule. SOS decisions based on MIBTA's feedback always selects the best performing schedule. This experiment proves that the accurate estimation of MIBTA can be used to improve system-level performance.

Finally, thread-progress-aware fetch policies such as the ones presented by Eyerman and Eeckhout [2009] could be applied. In this article the authors introduce a fetch policy that continuously monitors the progress of each thread in the SMT processor and gives more priority to the threads that are falling behind. Having a more accurate accounting mechanism such as MIBTA would increase the effectiveness of such a proposal.

6.2. Virtualized Environments

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

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

6.3. Dynamic Voltage and Frequency Scaling

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

6.4. Parallel Tasks

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

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

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

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

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

6.5. Other Proposals Providing Fairness

Several hardware approaches deal with the problem of providing fairness in multicore/multithreaded architectures. Although fairness is a desirable characteristic of a system, it has been shown that it cannot be used to provide an accurate CPU accounting [Luque et al. 2009].

Several proposals approach fairness in MT processors by providing the *same amount of resources* to each running task. However, ensuring a fixed amount of resources to a task does not translate into a direct CPU utilization that can be accounted to that task [Cazorla et al. 2005; Iyer et al. 2007; Nesbit et al. 2007; Raasch and Reinhardt 2003]. This is mainly due to the fact that the relation between the amount of resources assigned to a task and its performance can be different for each task. Hence, although all N tasks running in an MT processor receive 1/N of the resources, their relative progress is different, and so it should be their accounted CPU utilization.

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

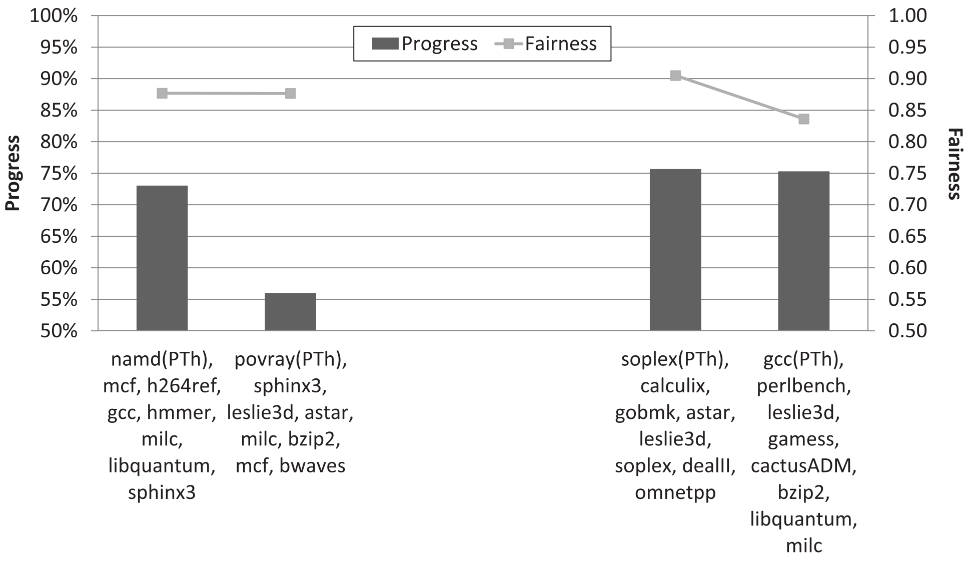

Fig. 12. Progress and fairness for four workloads in a 4-core 2-way SMT processor.

Hence, even if an architecture is known to be fair, it is not enough to properly account CPU utilization to each task.

For instance, in Figure 12, we show the fairness according to the second flavor of fairness explained above, and the progress of different tasks in a workload. In particular, we compute the progress of the Principal Thread (PTh) and its fairness in four workloads running on a 4-core processor in which each core is a 2-way SMT. Fairness is measured as $1 - (\frac{\sum_{k=1}^{N} |P_k - P_{avg}|}{N})$, where N is the number of benchmarks in the workload, P_k is the progress of a task K, and P_{avg} is the average progress in the workload. We observe that in the two workloads on the left, *(namd, mcf, h264ref, gcc, hmmer, milc, libquantum, sphinx3)* and *(povray, sphinx3, leslie3d, astar, milc, bzip2, mcf, bwaves)*, the progress of the PThs is quite different, while their fairness value is the same. In contrast, for the workloads on the right, *(soplex, calculix, gobmk, astar, leslie3d, soplex, dealII, omnetpp)* and *(gcc, perlbench, leslie3d, gamess, cactusADM, bzip2, libquantum, milc)*, the fairness is significantly different whereas the PThs have roughly the same progress.

6.6. Scalability

In terms of performance degradation and hardware cost, the worst situation for MIBTA is when all tasks that run in each hardware thread are independent. Under this scenario we have shown that with performance degradations between 1.0% and 3.2%, and reduced hardware budget, MIBTA provides better accounting accuracy than previous accounting approaches.

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

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

7. RELATED WORK

We are not aware of any other work that studies CPU accounting for CMP+SMT architectures. Several proposals seek to provide *quality of service* in MT processors [Cazorla et al. 2006; Iyer et al. 2007; Guo et al. 2007; Qureshi and Patt 2006; Nesbit et al. 2008, 2007; Yeh and Reinman 2005]. They guarantee an amount of resources to each task to ensure performance isolation through some hardware mechanisms on CMP or SMT processors. Similarly to MIBTA, two of these approaches [Cazorla et al. 2006; Yeh and Reinman 2005] split the execution of tasks in isolation and multithreaded phases, but they are not aware of inter-task conflicts during the isolation phase, and cannot provide an accurate estimate of task's performance in isolation.

However, ensuring a fixed amount of resources to a task does not translate into a CPU utilization that can be computed to that task. The main reason behind this is that the relation between the amount of resources assigned to a task and its performance is different for each task. In addition, even if a task is reserved a fixed amount of processor resources, like the ROB, the issue queues, fetch priority, etc., this task may suffer more than 36% variation in its performance depending on the other tasks it is run with in the MT architecture [Cazorla et al. 2005].

In other approaches [Ebrahimi et al. 2010; Mutlu and Moscibroda 2007] for CMP processors, the goal is to provide fairness in the shared memory system through control of the number of memory requests from each different task. These proposals achieve the same interference for each task in the memory hierarchy. However, this is different from our approach, since we seek to measure the effect of inter-task interferences on the progress of each task.

Fedorova et al. [2007] propose a cache-aware scheduler that compensates tasks that have high cache contention by giving them extra CPU time. Other authors propose to reduce cache interference by combining tasks with different characteristics [Dhiman et al. 2009; Knauerhase et al. 2008]. However, these works do not know the interference in on-core shared resources.

8. CONCLUSIONS

This article demonstrates that the current accounting mechanisms are not as accurate as they should be in CMP+SMT processors. Though these mechanisms enhance the accuracy of accounting in one of the existing TLP paradigms, they do not consider the interaction of several TLP paradigms at the same time; hence they introduce inaccuracies in the CPU time accounting. To solve this problem, we introduce a new accounting mechanism which does not depend on the combination of TLP paradigms implemented in the target architecture. We call this mechanism *Micro-Isolation-Based Time Accounting* (MIBTA). MIBTA reduces the off estimation under 5.0% in average on an 8-core 2-way SMT processor, while previous proposals present average off estimations over 21%. In addition, we develop a new inter-task misses tracking mechanism called *randomized sampled ATD* (RSA). RSA decreases the ATD overhead to 960 bytes in a 2MB LLC cache, while preserving its high accuracy. At core level MIBTA does not track every hardware shared resource, reducing its implementation cost to the minimum.

ACKNOWLEDGMENTS

The authors want to thank the anonymous reviewers for their insightful feedback on this work.

REFERENCES

ACOSTA, C., CAZORLA, F. J., RAMIREZ, A., AND VALERO, M. 2009. The MPsim simulation tool. http://capinfo.e.ac.upc.edu/PDFs/dir21/file003472.pdf.

AGARWAL, A., LIM, B. -H., KRANZ, D.,KUBIATOWICZ, J. 1991. A processor architecture for multiprocessing. Tech. rep. MIT/LCS/TM-450, MIT, http://www.dtic.mil/dtic/tr/fulltext/u2/a237476.pdf.

BEKER, H. AND PIPER, F. 1982. *Cipher Systems: The Protection of Communications.* Wiley-Interscience.

BROYLES, M., FRANCOIS, C., GEISSLER, A., HOLLINGER, M., ROSEDAHL, T., ET AL. 2011. IBM energyscale for POWER7 processor-based systems. http://www.ibm.com/systems/power/hardware/whitepapers/energyscale7.html.

CAZORLA, F. J. FERNANDEZ, E., GRAN, D., SPAIN, C., RAMIRIZ, A. ET AL. 2005. Architectural support for real-time task scheduling in SMT systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES).*

DHIMAN, G., MARCHETTI, G., AND ROSING, T. 2009. Vgreen: A system for energy efficient computing in virtualized environments. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED'09).*

EBRAHIMI, E., LEE, C. L., MUTLU, O., AND PATT, Y. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10).*

EYERMAN, I., EECKHOUT, L., KARKHANIS, T., AND SMITH, J. 2006. A performance counter architecture for computing accurate cpi components. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06).*

EYERMAN, S. AND EECKHOUT, L. 2009. Per-Thread cycle accounting in smt processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09).*

FEDOROVA, A., SELTZER, M., AND SMITH, M. D. 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'07).*

GIBBS, B., PULLES, M., ET AL. 2006. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbook.

GUO, F., SOLIHIN, Y., ZHAO, L., AND IYER, R. 2007. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07).*

HALSTEAD, R. AND FUJITA, T. 1988. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the International Symposium on Computer Architecture (ISCA'88).*

ITRS. 2011. International technology roadmap for semiconductors. http://www.itrs.net.

IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENEI, S., ET AL. 2007. QoS policies and architecture for cache memory in cmp platforms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurements and Modeling of Computer Systems.*

KNAUERHASE, R. C., BRETT, P., HOHLT, B., LI, T., AND HAHN, S. 2008. Using os observations to improve performance in multicore systems. *IEEE Micro 28*, 3, 54–66.

LUQUE, C., MORETO, M., CAZORLA, F. J., GIOIOSA, R., BUYUKTOSUNOGLU, A., AND VALERO, M. 2009. ITCA: Inter-Task conflict-aware cpu accounting for cmps. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'09).*

LUQUE, C., MORETO, M., CAZORLA, F. J., GIOIOSA, R., BUYUKTOSUNOGLU, A., AND VALERO, M. 2011. CPU accounting for multicore processors. *IEEE Trans. Comput. 61*, 251–264.

MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2, 78–117.

MUTLU, O. AND MOSCIBRODA, T. 2007. Stall-Time fair memory access scheduling for chip multiprocessors. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07).*

NESBIT, K. J., LAUDON, J., AND SMITH, J. E. 2007. Virtual private caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07).*

NESBIT, K. J., MORETO, M., CAZORLA, F. J., RAMIREZ, A., VALERO, M., AND SMITH, J. E. 2008. Multicore resource sharing. *IEEE Micro 28*, 6–16.

OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. 1996. The case for a single-chip multiprocessor. *SIGPLAN Not. 31.*

ORACLE. 2012. White paper. Oracle's sparc t4-1, sparc t4-2, sparc t4-4, and sparc t4-1b server architecture. http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o1.

QURESHI, M. K. AND PATT, Y. N. 2006. Utility-Based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'06).*

RAASCH, S. E. AND REINHARDT, S. K. 2003. The impact of resource partitioning on smt processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'03).*

ROTEM, E., NAVEH, A., RAJWAN, D., ANANTHAKRISHNAN, A., AND WEISSMANN, E. 2011. Power management architecture of the 2^{nd} generation intel core microarchitecture, formerly codenamed sandy bridge. In *Proceedings of the Symposium on High Performance Chips (HotChips'11).*

SERRANO, M. J., WOOD, R., AND NEMIROVSKY, M. 1993. A study on multistreamed superscalar processors. Tech. rep. 93-05, University of California Santa Barbara.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'01).*

SINHAROY, B., KALLA, R., STARKE, W. J., LE, H. Q., CARGNONI, R., VAN NORSTRAND, J. A., RONCHETTI, B. J., STUECHELI, J., LEENSTRA, J., GUTHRIE, G. L., NGUYEN, D. Q., BLANER, B., MARINO, C. F., RETTER, E., AND WILLIAMS, P. 2011. IBM POWER7 multicore server processor. *IBM J. Res. Devel. 55*, 191–219.

SMITH, B. 1981. Architecture and applications of the hep multiprocessor computer system. In *Proceedings of the 4^{th} Symposium on Real Time Signal Processing.*

SNAVELY, A. AND TULLSEN, D. M. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00).*

SNAVELY, A., TULLSEN, D. M., AND VOELKER, G. 2002. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

STORINO, S., AIPPERSPACH, A., BORKENHAGEN, J., EICKEMEYER, R., KUNKEL, S., LEVENSTEIN, S., AND UHLMANN, G. 1988. A commercial multithreaded risc processor. In *Proceedings of the 45^{th} International Solid-State Circuits Conference.*

TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA'95).*

YEH, T. Y. AND REINMAN, G. 2005. Fast and fair: Data-Stream quality of service. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05).*

Received June 2012; revised November 2012; accepted November 2012