Fairness Enforcement in Switch On Event Multithreading

RON GABOR

Tel Aviv University; Intel Corporation

SHLOMO WEISS

Tel Aviv University

and

AVI MENDELSON

Intel Corporation

The need to reduce power and complexity will increase the interest in Switch On Event multithreading (coarse-grained multithreading). Switch On Event multithreading is a low-power and low-complexity mechanism to improve processor throughput by switching threads on execution stalls. Fairness may, however, become a problem in a multithreaded processor. Unless fairness is properly handled, some threads may starve while others consume all of the processor cycles. Heuristics that were devised in order to improve fairness in simultaneous multithreading are not applicable to Switch On Event multithreading. This paper defines the fairness metric using the ratio of the individual threads' speedups and shows how it can be enforced in Switch On Event multithreading. Fairness is controlled by forcing additional thread switch points. These switch points are determined dynamically by runtime estimation of the single threaded performance of each of the individual threads. We analyze the impact of the fairness enforcement mechanism on aggregate IPC and weighted speedup. We present simulation results of the performance of Switch On Event multithreading. Switch On Event multithreading achieves an average aggregate IPC increase of 26% over single thread and 12% weighted speedup when no fairness is enforced. In this case, a sixth of our runs resulted in poor fairness in which one thread ran extremely slowly (10 to 100 times slower than its single-thread performance), while the other thread's performance was hardly affected. By using the proposed mechanism, we can guarantee fairness at different levels of strictness and, in most cases, even improve the weighted speedup.

Categories and Subject Descriptors: C.1.2 [Processor Architecture]: Multiple Data Stream Architectures (Multiprocessors)

A preliminary version of this paper entitled "Fairness and Throughput in Switch on Event Multithreading" appeared in the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006 (Micro-39) [Gabor et al. 2006].

Authors' addresses: Ron Gabor, Department of Electrical Engineering, Tel Aviv University, Tel Aviv; Intel Corporation, email: rongabor@post.tau.ac.il or ron.gabor@intel.com; Shlomo Weiss, Department of Electrical Engineering, Tel Aviv University, Tel Aviv; email: weiss@eng.tau.ac.il; Avi Mendelson, Intel Corporation; email: avi.mendelson@intel.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1544-3566/2007/09-ART15 \$5.00 DOI 10.1145/1275937.1275939 <http://doi.acm.org/10.1145/1275937.1275939>

General Terms: Design, Performance

Additional Key Words and Phrases: Switch on Event multithreading, SOE, coarse-grained multithreading, multithreading, fairness, performance, throughput, weighted speedup

ACM Reference Format:

Gabor, R., Weiss, S., and Mendelson, A. 2007. Fairness enforcement in switch on event multithreading. *ACM Trans. Archit. Code Optim.* 4, 2, Article 15 (September 2007), 34 pages. DOI = 10.1145/1275937.1275939 <http://doi.acm.org/10.1145/1275937.1275939>

1. INTRODUCTION

During the last two decades, different architectures were introduced to support multiple threads on a single die (chip). These architectures can be classified into three classes:

- Chip multiprocessor (CMP)—multiple processors (on die) that share some of the memory hierarchy, e.g., IBM’s Power4 [Tendler et al. 2002] and Intel Duo [Gochman et al. 2006].
- Simultaneous multithreading (SMT) in which instructions from multiple threads are fetched, executed, and retired on each cycle, sharing most of the resources in the core [Tullsen et al. 1996, 1998], e.g., IBM’s Power5 [Kalla et al. 2004] and Intel’s P4 [Marr et al. 2002].
- Switch On Event (SOE, coarse-grained multithreading) in which instructions from a single thread are fetched, executed, and retired, while an event, such as a long latency memory operation event is used to efficiently switch between the different threads [Eickemeyer et al. 1997; Farrens and Pleszkun 1991; Thekkath and Eggers 1994], e.g., IBM’s RS64 IV [Borkenhagen et al. 2000] and Intel’s Montecito [McNairy and Bhatia 2005].

A conclusive comparison of these architectures is by no means a trivial task since it involves many design and implementation details and, therefore, is out of the scope of this paper. In general, SOE is simple to implement and can easily be extended to a high number of threads. Not only does simpler implementation mean lower design effort [Bazeghi et al. 2005], but it usually also means lower power.

There is an ongoing trend toward lower power and complexity of microprocessors. All major microprocessor vendors are going to chip multiprocessors instead of increasing the processor complexity [Kalla et al. 2004; Krewell 2004; Rattner 2005] and many vendors use simple cores rather than complex superscalars [Kahle 2005; Kongetira et al. 2005]. In order to improve throughput and power efficiency, more cores and threads are squeezed into a single processor die [Cai 2000; Davis et al. 2005; Spracklen and Abraham 2005]. Asymmetric cores integration can further improve power to performance ratio by integrating simple cores with larger, more complex ones [Balakrishnan et al. 2005; Kumar et al. 2005]. SOE is extremely important to this “simple-cores” trend, as it can increase the number of threads at a relatively small power, area, and complexity costs. Given SOE’s importance, we feel it deserves further research and study.

Fig. 1. Intuitive example of unfair execution in SOE. *Ex1* marks execution of instructions from thread 1, *Ex2* from thread 2, *M* marks last-level cache misses, and *Sw* denotes thread switch overheads. When both threads run together using SOE (bottom), most of the execution cycles go to thread 1. The 2nd thread runs extremely slowly, while the 1st thread's performance is hardly affected by the multithreading.

In all the current multithreaded architectures, fairness is a major problem. Lack of fairness is usually caused when resources are unfairly shared among the different threads. Unfair execution can cause, for example, serious responsiveness problems, in which some threads run extremely slowly. CMP-based architectures are mainly exposed to fairness in accessing the shared caches in the memory hierarchy. SMT has to handle fairness among most of the resources of the machine and requires substantial changes in the microarchitecture, using ad-hoc techniques and heuristics. Fairness in SOE, as will be shown in this paper, can be handled with minimal intrusion into the microarchitecture.

1.1 Example 1: Fairness Problem

The following example demonstrates the fairness problem in SOE. Consider the simple two-thread case illustrated in Figure 1. The example shows how each of the two threads is executing by itself on the processor, as well as how they run together, using SOE, which switches threads on last-level cache misses. Let us assume one thread (thread 2) has many more last-level cache misses than the other (thread 1). When executed alone on the processor, each of the two threads will suffer from a certain stall for each miss, as it has nothing to execute while the miss is being resolved (external memory access latency). When executed together in SOE, however, these thread stalls are used for execution of instructions from the other thread. From each thread's perspective, each of these execution stalls ends when the other thread encounters a miss. This means that, although in the case of single thread, miss latency is effectively constant (memory access latency), in SOE each thread sees a different miss latency whose length is determined by the other thread's execution. This causes, in our scenario, for the slower thread to become much slower, while the faster thread gets most of the execution cycles. In this example, SOE improved throughput but caused unfair execution.

1.2 SOE and SMT

SOE maintains the context of a single thread. SMT and fine-grain multithreading, in general, is required to keep and update multiple thread contexts.

As a result, compared to SMT, SOE has lower complexity overhead and its implementation affects fewer units in the processor. Tune et al. [2004] mention some of the complexities, imposed by SMT, such as timing issues (increasing cycle time) and additional pipeline stages. Two additional pipeline stages were also added by Tullsen et al. [1996] to read and write the larger register file required by SMT. SOE does not require additional pipeline states or increase in register-file size.

SMT is pervasive. Its implementation affects a large number of units in the processor. SOE, on the other hand, requires fewer changes to an existing design. For example, there is no need for features required by SMT, such as a larger register file and a separate return stack, renaming table, retirement, and instruction flush for each thread [Tullsen et al. 1996].

Area overhead of SMT is larger than that of SOE. Implementation of SMT with small-area overhead is possible, but limits the performance gain. In order to achieve good performance from SMT a high area increase is required. An example of such an increase is IBM's Power5 core, which was about 24% larger than the Power4 core because of the addition of SMT [Glaskowsky 2003]. It should be noted that both SOE and SMT require some resources to be thread-tagged (e.g., TLBs, branch history), however, other changes imposed by SMT (e.g., duplicated logic, flush mechanism) are not required by SOE.

Fairness enforcement in SOE, as shown in this paper, is not microarchitecture dependent, scales easily to more than two threads and requires only a few counters in order to guarantee fairness. Dealing with fairness in SMT, on the other hand, relies on heuristics and ad-hoc techniques that collectively give a notion of fairness. For example, Raasch and Reinhardt [2003] showed that resource partitioning helps assuring that threads do not interfere with each other. This partitioning has a negative effect on SMT performance compared to SOE, where partitioning is not required. Partitioning also limits the scalability of the number of threads. In SMT, any shared resource requires some sort of mechanism to prevent starvation. Most of these mechanisms are microarchitecture dependent and are tailor made for each design (ad-hoc solutions). Many of these resources are not shared in SOE, hence, require less efforts in fairness enforcement (e.g., reservation stations, execution units, and return stack buffer). Having more than two threads can further complicate dealing with fairness in SMT (e.g., partitioning).

1.3 Related Work

Many studies were made on SOE and its variants (coarse-grained multithreading). Most of these studies dealt with throughput improvements. Farrens and Pleszkun [1991] introduced blocked multithreading (BMT) as part of their evaluation of techniques for improving processor throughput. In BMT, the active thread is switched off whenever it is blocked. Gupta et al. [1991] evaluated coarse-grain multithreading as part of their latency reducing and tolerating techniques. They evaluated it along with coherent caches, relaxed-memory consistency models, and prefetching techniques. Eickemeyer et al. [1997] studied SOE throughput in server workloads, and showed that SOE achieves its

maximum throughput using three threads. Haskins and Skadron [2001] introduced differential multithreading (dMT), a variant of BMT, which also handles misses in instructions and data caches. They showed that dMT can substantially reduce the cost and complexity of microprocessors. A complexity-related study proposed to add SOE on top of SMT in order to increase the number of threads with low-complexity overhead [Tune et al. 2004]. None of these studies dealt with fairness between threads. Our approach can be applied to any SOE mechanism, such as BMT or dMT, to improve execution fairness.

Coarse-grained multithreading (or SOE) has been implemented in several commercial processors, such as IBM's RS64 IV [Borkenhagen et al. 2000] and Intel's Montecito [McNairy and Bhatia 2005]. Montecito preferred SOE over SMT because of the already high IPC (instructions per cycle) and execution units utilization achieved in single-thread runs, which implies low potential for SMT. In Montecito's SOE scheme, each thread gets its fair share of the memory hierarchy caches. There is, however, no guarantee for fairness in its threads execution. Several research projects studied SOE multithreading [Agarwal et al. 1995, 1990; Gruenewald and Ungerer 1997; Mowry and Ramkisson 2000], but none of them dealt with the fairness problem. A survey of SOE research projects and commercial machines can be found in Ungerer et al. [2003] explicit multithreading survey.

Fair cache sharing among multiple coscheduled threads has been shown to be a potential cause of serious problems, such as threads starvation. Cache sharing can be extremely unfair, for example, when a thread with high miss rate and poor locality constantly causes evictions of other thread's data that will be required soon after. Dynamic and static resources partitioning schemes have been proposed to improve fairness in caches and other resources sharing [Cazorla et al. 2004; Kim et al. 2004]. Kim et al. [2004] studied fairness issues in cache sharing in CMP. They showed that optimizing for fairness also increases throughput while maximizing throughput does not necessarily improve fairness. In SOE, as shown in our study, fairness enforcement has limited effect on the IPC when the two threads have comparable performance. Enforcing fairness for dissimilar threads requires switches, which tend to lower the aggregate IPC, but, in most cases, improve the weighted speedup.

Simple time sharing is used at the operating system level to ensure an equal share of time for each thread. Various methods were suggested to manage the time sharing for fairness with prioritization and real-time constraints [Duda and Cheriton 1999; Jones et al. 1997]. We deal with the applicability of time sharing to SOE in example 3 (Section 2).

Fairness of threads execution was studied in the context of SMT [Cazorla et al. 2004; Luo et al. 2001; Raasch and Reinhardt 1999, 2003; Tullsen and Brown 2001]. Raasch and Reinhardt [2003] showed that resource partitioning in SMT improves threads' execution fairness. For example, statically partitioned ROB improves fairness compared to competitive sharing. Luo et al. [2001] used fetch policy as a heuristic to prioritize the different threads, in order to improve fairness. Both approaches are applicable to SMT, but not to SOE. SOE maintains a single active thread in the pipeline. Hence, resource partitioning will not improve fairness and fine-grained fetch prioritization will

require frequent pipeline flushes in order to switch threads (severely harming performance).

Luo et al. [2001] suggested a single metric that combines both fairness and throughput. They defined fairness as the harmonic mean of the speedups of the individual threads. The speedup they use is the throughput (*IPC*) of each individual thread when run in multithreaded mode, compared to its throughput when executed alone on the processor. They attempt to capture both throughput and fairness in a single metric. We use the harmonic mean metric to analyze our results in Section 5.6.

Snively and Tullsen [2000] used weighted speedup in order to measure the goodness of their job scheduling. Weighted speedup is the sum of the speedups of individual threads.¹ Weighted speedup is used to fairly measure relative throughput. Using only throughput (aggregate *IPC*) as a measure of goodness may be inadequate and misleading as it fails to show how each thread was affected [Sazeides and Juan 2001]. Weighted speedup assures that high throughput threads are not preferred over lower throughput threads. We use both aggregate *IPC* and weighted speedup to gain insight into the operation of our mechanism.

1.4 Paper Overview

No attempt has been made so far in the relevant literature to analyze or control fairness in SOE multithreading. This paper fills this gap. We provide a mechanism for fairness enforcement and analyze it. The suggested mechanism uses hardware counters and a feedback loop that monitors fairness by estimating the *IPC* of individual threads, had each of them been executed alone on the processor. Fairness between threads is enforced by inducing additional thread switches in order to balance threads execution.

For simplicity, the rest of the paper uses last-level cache misses as the event that causes thread switches. The suggested approach is applicable to any detectable long latency stall. This paper makes the following contributions:

- *Analytical Model.* We provide an analytical model and analyze SOE fairness and throughput. The analytical model shows the effects of the induced thread switches. It enables throughput calculation given workload characteristics such as miss distribution and threads performance when executed alone on the processor. The main benefit of the model is the estimation method, which is essential for enforcing fairness.
- *Fairness Enforcement in SOE.* We present a low overhead mechanism for fairness enforcement in SOE multithreading. The mechanism tracks runtime fairness by estimating the individual threads' performance, had they been executed alone on the processor. Thread switches are induced, when necessary, in order to enforce the desired fairness level.

The rest of the paper is organized as follows. Section 2 describes the fairness-enforcement mechanism and analyzes it using an analytical model. Section 3

¹Weighted speedup is defined as $WS = \sum_{i=1}^N (\text{realized IPC job}_i / \text{single-threaded IPC job}_i)$.

Table I. Abbreviations and Symbols

Symbol	Description
IPC	Average number of useful instructions executed per cycle.
IPC_{no_miss}	IPC of a thread excluding last-level misses (as if the last level cache always hits).
IPC^{SOE}	IPC (of all threads) when executed using SOE.
IPC_j^{SOE}	IPC of thread j when executed using SOE with other threads.
IPC_j^{ST}	IPC of thread j when executed alone.
IPM_j	Instructions per miss in thread j . Average number of instructions between two consecutive misses in thread j .
CPM_j	Cycles per miss in thread j . Average number of cycles between two consecutive misses in a thread j .
CPM_{min}	Minimal CPM_j of all threads ($\min_j CPM_j$).
$IPSw_j$	Instructions Per Switch in thread j . Average number of instructions thread j executes before it is switched out (in SOE).
$CPSw_j$	Cycles per switch in thread j . Average number of cycles thread j executes before it is switched out (in SOE).
WS	Weighted speedup: $WS = \sum_{j=1}^N (IPC_j^{SOE} / IPC_j^{ST})$
$\mathbb{F}$	Fairness to be enforced ($0 \leq \mathbb{F} \leq 1$).
λ	The period at which fairness enforcement parameters are recalculated (in cycles).

presents implementation issues of the proposed mechanism. Section 4 describes simulation tools and methodology including machine configuration and SOE implementation details. Sections 5 and 6 present simulation results for a two-thread SOE machine and discuss them. Conclusions are presented in Section 7.

2. MODEL AND FAIRNESS DEFINITION

This section presents an analytical model for SOE fairness and throughput. The model provides a fairness-estimation method, which is used by the mechanism for fairness enforcement at runtime.

Sketching the mechanism roughly, we estimate the single-thread performance of the individual threads had each of them been executed alone, while they are running in SOE multithreading. The estimation is based on the measurement of the throughput of each thread while it is running under SOE, in addition to last-level cache misses, which would have stalled the thread, had it been executed alone. We can then estimate the speedup of each thread by dividing its actual SOE throughput by the estimated single-thread throughput. Our proposed mechanism induces additional thread switches in order to make sure that the speedups are similar for all of the threads. We define fairness as the ratio between the speedup of the individual threads. We compute the necessary “instructions per switch” quota that needs to be maintained in order to guarantee a minimum specified level of fairness. This instructions quota is then maintained using deficit counting (as explained in Section 3.2).

Fig. 2. Representation of single thread runs for two threads (top) and SOE run of the same two threads (bottom). IPM_j and CPM_j are the average number of instructions and cycles between two misses in thread j (excluding miss handling cycles).

2.1 Program Behavior Model

In order to present a simple and meaningful analytical model of the fairness problem, let us assume that the execution of single-thread applications can be viewed as a sequence of instructions delimited by long-latency last-level cache misses. Let IPM (instructions per miss) be the average number of useful instructions executed between two consecutive misses and CPM (cycles per miss) be the average number of cycles between those misses. When a thread executes alone on the processor, each miss causes an execution stall of $Miss_{latency}$ cycles ($Miss_{latency}$ is the average memory access time in processor cycles). IPM_j and CPM_j (IPM and CPM of thread j) are illustrated in Figure 2 for two threads both when running together (using SOE), and when executed alone.

Let IPC_j^{ST} be the average number of useful instructions executed per cycle (retired instructions per cycle) in thread j when executed alone on the processor. As shown in Eq. (1), IPC_j^{ST} can be calculated by dividing IPM_j , by the average of the total number of cycles corresponding to these instructions,² which is $CPM_j + Miss_{latency}$.

$$IPC_j^{ST} = \frac{IPM_j}{CPM_j + Miss_{latency}} \quad (1)$$

Let IPC_j^{SOE} be the average number of useful instructions executed per cycle in thread j when running together with other threads using SOE. As shown in Eq. (2), IPC_j^{SOE} can be calculated by dividing IPM_j by the average total

²We deliberately ignore out-of-order issues, such as useful work done, while cache misses are being resolved or the fact that several cache misses may be pending (overlapping of cache misses). This is done in order to simplify the model. We will further deal with these issues later in the paper (using empirical results).

number of cycles of a whole switch-on-event round (until thread j gains execution again). A whole round is calculated by summing the CPM of all the threads together with their corresponding $Switch_{latency}$ cycles (the average overhead per thread switch).

$$IPC_j^{SOE} = \frac{IPM_j}{\sum_k (CPM_k + Switch_{latency})} \quad (2)$$

It should be noted that Eq. (2) holds as long as misses that cause thread switches are resolved by the time their threads are running again. This is obviously true if there are sufficient threads available.

It should be noted that Eq. (2) holds as long as misses that cause thread switches are resolved by the time their threads are running again. This is obviously true if there are sufficient threads available. It should also be noted that we assume that round robin is used for switching threads. The round-robin assumption means that all the threads are used in each round. This comes from practical aspects of the number of threads used for SOE. The number of threads should be the minimal number that is enough to hide the stalls. This number should be minimal in order to reduce resource contentions (e.g., competition over TLBs or any other shared resource).

2.2 Fairness Metric

A system is fair if all the threads experience an equal slowdown compared to their performance had they been executed alone [Luo et al. 2001]. We define a perfectly fair system as:

$$\forall j, k \quad \frac{IPC_j^{SOE}}{IPC_j^{ST}} = \frac{IPC_k^{SOE}}{IPC_k^{ST}} \quad (3)$$

Following Eq. (3), a fairness metric can be defined as the minimum ratio between the slowdowns of any two threads running in the system:

$$Fairness \equiv \frac{\min_j \left(\frac{IPC_j^{SOE}}{IPC_j^{ST}} \right)}{\max_k \left(\frac{IPC_k^{SOE}}{IPC_k^{ST}} \right)} = \min_{j,k} \left(\frac{\left(\frac{IPC_j^{SOE}}{IPC_j^{ST}} \right)}{\left(\frac{IPC_k^{SOE}}{IPC_k^{ST}} \right)} \right) = \min_{j,k} \left(\frac{IPC_j^{SOE} \cdot IPC_k^{ST}}{IPC_j^{ST} \cdot IPC_k^{SOE}} \right) \quad (4)$$

We use “min” to denote the minimal element (nothing is smaller) of a set and “max” to denote the maximal element. It should be noted that this definition of fairness is more strict than the harmonic mean used by Luo et al. [2001]. It uses the maximum and minimum speedups for fairness calculation, which guarantees that these values will not have reduced impact on fairness because of averaging. In other words, enforcing fairness using our definition is guaranteed to improve harmonic mean-based fairness, but not vice versa.

2.3 Fairness Enforcement

According to the definition in Eq. (4), $0 \leq Fairness \leq 1$. Perfect fairness is achieved when $Fairness = 1$. Fairness decreases with the metric value, until

it reaches 0 which is complete unfairness (one of the threads is completely starved—not running at all).

Substituting Eq. (1) and (2) into (4) results in Eq. (5):

$$Fairness \equiv \min_{j,k} \left(\frac{CPM_j + Miss_{latency}}{CPM_k + Miss_{latency}} \right) \quad (5)$$

Equation (5) shows that unless we modify our SOE scheme, fairness will be defined by CPM , which is a characteristic of the running threads.

In order to control fairness in SOE, we must define forced switch points, not necessarily caused by last-level cache misses. Let $IPSw_j$ be the average number of instructions executed from thread j , before it is switched out. Similarly, let $CPSw_j$ be the average number of cycles thread j has executed before it is switched out (if thread j switches out only because of the last-level cache misses, then $IPSw_j = IPM_j$ and $CPSw_j = CPM_j$). Using $IPSw_j$ and $CPSw_j$ modifies Eq. (2):

$$IPC_j^{SOE} = \frac{IPSw_j}{\sum_k (CPSw_k + Switch_{latency})} \quad (6)$$

Substituting Eqs. (6) and (1) into Eq. (4), we get:

$$Fairness \equiv \min_{j,k} \left(\frac{IPSw_j IPM_k (CPM_j + Miss_{latency})}{IPSw_k IPM_j (CPM_k + Miss_{latency})} \right) \quad (7)$$

We define the parameter $\mathbb{F}$ to be the target fairness that we wish to achieve from the system ($0 \leq \mathbb{F} \leq 1$). Based on our definition of fairness from Eq. (7), a system achieves the required fairness $\mathbb{F}$ if it satisfies Eq. (8).

$$\mathbb{F} \leq \min_{j,k} \left(\frac{IPSw_j IPM_k (CPM_j + Miss_{latency})}{IPSw_k IPM_j (CPM_k + Miss_{latency})} \right) \quad (8)$$

Let CPM_{min} be the minimal value of CPM of all threads, $CPM_{min} = \min_j CPM_j$. Setting $IPSw_j$ for each thread, as shown in Eq. (9) is guaranteed to satisfy Eq. (8).³

$$IPSw_j = \min \left(IPM_j, \frac{IPC_j^{ST}}{\mathbb{F}} (CPM_{min} + Miss_{latency}) \right) \quad (9)$$

Equation (9) is the main result of the analytical model. It is used by the fairness-enforcement mechanism in order to calculate $IPSw_j$ for each thread that, when enforced, will achieve the required fairness.

Our SOE scheme switches threads in order to achieve an average of $IPSw_j$ instructions per switch. It switches on last-level cache misses in addition to forced switches because of $IPSw_j$. Hence, there is no way to increase $IPSw_j$ to a value greater than IPM_j (that is why we use min in Eq. (9)).

As shown in Eq. (4), fairness is the minimum ratio of speedups between any two threads in the system. When used as a parameter $\mathbb{F}$, it sets the allowed speedup ratio. For example, calculating $IPSw_j$ for all threads using $\mathbb{F} = \frac{1}{2}$ will guarantee a maximum speedup ratio of 2. This means that the ratio of speedups

³This can be proved algebraically.

between the threads that have the highest and lowest speedup will be, at most, 2.

2.4 Fairness and Throughput

Using $IPSw_j$ to control fairness sets new thread switches. These switches are defined by the $IPSw_j$ quota and not necessarily by last-level cache misses. In other words, there are forced-thread switches, that cause some execution stall (thread switch latency), which are not hiding any other long latency stall (such as the last-level cache miss). This means that fairness enforcement will affect the throughput.

We can measure IPC^{SOE} , which is the aggregate IPC of all of the threads when executed together using SOE. As shown in Eq. (10), IPC^{SOE} can be calculated by dividing the total average number of instructions executed by all threads in a single SOE threads round, by the respective number of cycles. Finally, Eq. (6) can be used to show that $IPC^{SOE} = \sum_k IPC_k^{SOE}$:

$$IPC^{SOE} = \frac{\sum_k IPSw_k}{\sum_k (CPSw_k + Switch_{latency})} = \sum_k IPC_k^{SOE} \quad (10)$$

Another measure we use is weighted speedup (WS), defined as the sum of the speedups of the individual threads:

$$WS = \sum_{j=1}^N (IPC_j^{SOE} / IPC_j^{ST}) \quad (11)$$

2.5 Example 2: Fairness Enforcement

This example demonstrates the use of fairness enforcement using the method described in Section 2.3. Consider the case of two threads sharing a processor using SOE, switching on last-level cache miss events. Let us assume that instructions run at a rate of 2.5 instructions per cycle on both threads, excluding miss stalls ($IPC_{no_miss} = 2.5$). Memory access latency on a last-level cache miss is 300 cycles. Thread switch latency is 25 cycles. The first thread has a miss every 15,000 instructions (6,000 cycles) and the second every 1,000 instructions (400 cycles). Table II shows the performance of the two threads when running alone (IPC_j^{ST}) and when running together in SOE mode (IPC_j^{SOE}). As shown, in the latter case, the first thread's IPC drops by a factor of 1.02, while the other thread's IPC drops by 9.2. This is clearly unfair, as the faster thread (thread 1), whose performance is hardly affected, causes the other thread to slow down by a factor of 9. This gives a fairness metric of 0.11. However, when fairness is enforced to 1, the first thread is forced to switch every 1,667 instructions (on average), instead of only on cache misses, which occur every 15,000 instructions ($IPM_1 = 15,000$). This example shows that IPC^{SOE} (aggregate IPC) slightly drops when fairness is enforced, as a result of the additional thread switches and the diversion of execution toward the slower thread (the one with lower IPC_{no_miss}). Weighted speedup, however, shows that the fairness enforcement improves the overall throughput.

Table II. Example of SOE of Two Threads With and Without Fairness Enforcement^a

	$\mathbb{F} = 0$		$\mathbb{F} = 1$		$\mathbb{F} = \frac{1}{2}$	
	Th 1	Th 2	Th 1	Th 2	Th 1	Th 2
IPM_j	15,000	1,000	15,000	1,000	15,000	1,000
$IPC_{no,miss}$	2.5	2.5	2.5	2.5	2.5	2.5
$IPSw_j$	15,000	1,000	1,667	1,000	3,333	1,000
$CPSw_j$	6,000	400	667	400	1,333	400
IPC_j^{ST}	2.381	1.429	2.381	1.429	2.381	1.429
IPC_j^{SOE}	2.326	0.155	1.493	0.896	1.869	0.561
$\frac{IPC_j^{ST}}{IPC_j^{SOE}}$	1.02	9.22	1.59	1.59	1.27	2.55
IPC^{SOE} (Throughput)	2.48		2.39		2.43	
Achieved fairness	0.11		1.00		0.50	
Weighted speedup	1.09		1.25		1.18	

^aWhen no fairness is enforced ($\mathbb{F} = 0$), thread 2 is almost starved. When $\mathbb{F} = 1$ (fairness enforced to 1), both threads are forced to equal slowdown compared to their IPC_j^{ST} . When $\mathbb{F} = \frac{1}{2}$, slowdown is allowed to differ by, at most, a factor of 2.

2.6 Example 3: Simple Time Sharing

In this example, we analyze a simple time-sharing mechanism (for fairness improvement), which maintains a predefined constant average number of cycles in between switches.⁴ We use the same parameters as in Example 2, but with different $IPC_{no,miss}$ for each thread. Table III shows the SOE with no fairness enforcement ($\mathbb{F} = 0$), fairness enforcement using $\mathbb{F} = 1$, and simple time sharing using 5000 and 500 cycles as cycles per switch quota. It should be noted that it is not possible to run a thread for a time sharing quota that is higher than its average cycles between misses (as misses also cause thread switching). This is the case when trying to enforce a 5000 cycles quota on thread 2 (which switches, on average, every 1000 cycles, because of its misses), and the result is poor fairness (0.25). Even with a reasonable choice of the 500 cycles quota the achieved fairness (0.81) is still well below 1.00.

Figure 3 shows the achieved fairness as well as the weighted speedup. As shown, using a high cycles quota for time sharing (TS 5000) results in low achieved fairness, similar to when no fairness is enforced (regular SOE, $\mathbb{F} = 0$). When a low value is used for cycles quota (TS 500), reasonable fairness is achieved, but weighted speedup is still lower than that of using $\mathbb{F} = 1$ (fairness enforcement). It is not possible to tune the simple time-sharing mechanism for either fairness or weighted speedup. Its throughput and fairness achieved depend on the actual characteristics of the threads.⁵

2.7 Fairness and Throughput Analysis

When fairness is not enforced, every thread switch hides a memory access stall (lastlevel cache miss). However, when fairness is enforced, additional thread

⁴this can be done using deficit counting, as will be explained later in the paper.

⁵It should be noted that by using dynamic time-sharing, which is adaptive to the threads' characteristics, the weighted speedup can be improved.

Table III. Example of Two-Thread SOE with Fairness Enforcement and a Time Sharing Mechanism^a

	$\mathbb{F} = 0$		$\mathbb{F} = 1$		TS 5000		TS 500	
	Th 1	Th 2	Th 1	Th 2	Th 1	Th 2	Th 1	Th 2
IPM_j	15,000	1,000	15,000	1,000	15,000	1,000	15,000	1,000
IPC_{no_miss}	2.5	1.0	2.5	1.0	2.5	1.0	2.5	1.0
$IPSw_j$	15,000	1,000	3,095	1,000	12,500	1,000	1,250	500
$CPSw_j$	6,000	1,000	1,238	1,000	5,000	1,000	500	500
IPC_j^{ST}	2.381	0.769	2.381	0.769	2.381	0.769	2.381	0.769
IPC_j^{SOE}	2.128	0.142	1.353	0.437	2.066	0.165	1.190	0.476
$\frac{IPC_j^{ST}}{IPC_j^{SOE}}$	1.119	5.423	1.760	1.760	1.152	4.654	2.000	1.615
IPC^{SOE} (Throughput)	2.27		1.79		2.23		1.67	
Achieved fairness	0.21		1.00		0.25		0.81	
Weighted speedup	1.08		1.14		1.08		1.12	

^aTS n is time sharing using n cycles.

Fig. 3. Comparison of achieved fairness and weighted speedup for time sharing (example 3). TS n is time sharing using n cycles.

switches are induced in order to maintain the required fairness. Intuitively, these forced thread switches, which cause a $Switch_{latency}$ cycles stall, reduce the throughput of SOE.⁶

Figure 4a shows the effect of fairness enforcement on the aggregate IPC in a two-thread model. It shows IPC drop resulting from fairness enforcement, of thread pair combinations with different IPC_{no_miss} (performance excluding misses) and IPM . The figure shows that, in most cases, fairness enforcement causes an IPC drop. As shown by the lines of $IPC_{no_miss} = [2.5, 2.5]$, when IPC_{no_miss} is similar for both threads, IPC drops by up to 4%. However, when the

⁶It will be shown later that additional forced switches can actually increase the aggregate IPC.

(a) Fairness enforcement effect on aggregate IPC.

(b) Fairness enforcement effect on weighted speedup.

Fig. 4. The effect of fairness enforcement on aggregate IPC and weighted speedup, using a two-thread analytical model ($Miss_{latency} = 300$, $Switch_{latency} = 25$). Legend's notation: $IPC_{no_miss} = [IPC_{no_miss1}, IPC_{no_miss2}]$, $IPM = [IPM_1, IPM_2]$.

two threads do not have the same IPC_{no_miss} , IPC can drop by up to 15% or can actually improve by up to 10%. IPC drop is explained by the execution stalls induced by the additional switches, as well as the diversion of execution toward the thread with the lower IPC_{no_miss} . The IPC increase in the $IPC_{no_miss} = [2, 3]$ cases is explained by noting that fairness enforcement diverts the execution toward the thread with the higher IPC_{no_miss} and, hence, improves aggregate IPC.

Figure 4b shows the effect of fairness enforcement on the weighted speedup. It shows that weighted speedup improves even when additional thread switches are induced. This indicates that most of the aggregate IPC drop is because of the diversion of execution cycles to the slower thread (which improves the weighted speedup) and not by the additional thread switches (pipeline flushes) induced by the fairness enforcement mechanism. The analytical model shows that weighted speedup improves with the enforced fairness.

3. IMPLEMENTATION

In order to enforce fairness between threads, the processor should calculate $IPSw_j$ for each thread, based on its runtime characteristics, and regulate the switch points in order for the average instructions per switch to be equal to the required $IPSw_j$.

3.1 Runtime Calculation of $IPSw_j$

In order to calculate $IPSw_j$ (9), two thread characteristics must be obtained for each thread: IPM_j and CPM_j . In addition, $Miss_{latency}$ must be known. All of these can be obtained using three hardware counters per thread. These counters should count instructions retired, last-level cache misses, and cycles, while the threads are running in SOE:

Instrs_j: instructions retired from thread j .

Cycles_j: total number of cycles from the retirement of the first instruction after thread j switches in, until it is switched out (this is the actual number of cycles the thread was running for, excluding the switch overhead).

Misses_j: number of last-level cache misses encountered while running thread j . In order to minimize inaccuracies caused by overlapping of misses (several clustered pending misses on a specific thread caused by out-of-order execution), we count only misses that actually cause a thread switch (nonresolved misses that are encountered in the next-to-retire instruction).

The characteristics of the threads can be computed from these counters, as shown in Eqs.(12), (13), and (14). Average $Miss_{latency}$ can be either a predefined parameter or a measured statistic.

$$IPM_j = \frac{Instrs_j}{\max(Misses_j, 1)} \quad (12)$$

$$CPM_j = \frac{Cycles_j}{\max(Misses_j, 1)} \quad (13)$$

$$IPC_j^{ST} = \frac{IPM_j}{CPM_j + Miss_{latency}} \quad (14)$$

$IPSw_j$ can be calculated using the hardware counters every λ cycles. The calculated $IPSw_j$ values will be used during the following λ cycles. In other words, hardware counters of each λ cycles are used as an estimation for the

following λ cycles. λ should be set to a value large enough to allow good statistical averaging, but not too large in order to allow performance phases to be accurately tracked.

In rare cases, where a thread does not encounter any miss in λ cycles, a value of $Misses_j = 1$ is used to estimate its performance. This decreases IPC_j^{ST} estimation, however, it is still good enough for our purposes.

There are several alternative implementations for calculating $IPSw_j$ every λ cycles. It can be done in hardware, using injected instruction flows (flows of instructions that are injected into the pipeline by the hardware, as a result of an event) or using interrupts.

3.2 Maintaining $IPSw_j$ Using Deficit Counts

Fairness mechanism is expected to maintain $IPSw_j$ instructions, on average, between switches for thread j . However, simply forcing a thread switch every $IPSw_j$ instructions will not produce the expected average instructions per switch, as threads are switched on last-level cache misses as well. In order to achieve this average, we are using a per thread deficit counter, for dynamically adjusting the switch points. Deficit counters are used to hold the quota “left-over”. This leftover is caused by misses encountered before the quota is fully used up. The leftover increases the quota of the thread the next time it is switched in. This deficit mechanism is done in a similar manner to deficit-round-robin mechanism [Shreedhar and Varghese 1996].

The hardware maintains a per thread deficit counter. Initially, the deficit counter is set to 0 for all of the threads. Deficit counter of thread j is incremented by $IPSw_j$ every time thread j is switched in. On retirement of each instruction, the corresponding deficit counter is decremented. A thread is switched out when its deficit counter reaches 0, or when a last-level cache miss is encountered.

Deficit counting ensures that when a miss event causes a switch before $IPSw_j$ instruction quota was reached, the next instructions quota for that thread will be larger. This compensates for the shorter previous quota, that ended by a miss. The deficit mechanism ensures that, on average, the thread will run for the required $IPSw_j$ instructions between switches.

4. SIMULATION METHODOLOGY

An out-of-order processor was simulated using a detailed cycle accurate execution-driven simulator. The processor is derived from the P6 microarchitecture [Gwennap 1995]. The simulator uses long instruction traces (LITs) [Singhal et al. 2004]. LITs are not actually traces (they do not contain an instruction or execution trace). Each LIT contains an architectural checkpoint (state snapshot) in addition to injectable external events. Each checkpoint consists of a memory image and processor state (registers). Injectable external events include interrupts, IO, and DMA events. Using LIT enables full accurate simulation of the application, as well as of interrupts, operating system, and DMA side effects. These tools and methodology were extensively used for detailed simulation in many studies [Akkary et al. 2004; Cooksey et al. 2002; Falcon et al. 2004; Mutlu et al. 2003; Singhal et al. 2004].

Fig. 5. High-level view of processor pipeline and the memory hierarchy (top right).

Table IV. Simulated Machine Parameters

Fetch/retire width	4 / 4
RS size	36 entries
ROB size	96 entries
Load/store buffers	48 / 36 entries
L1 D-cache	32K, 8 ways, 64B line, 2 load ports, 1 store port
L1 I-cache	32K, 4 ways, 64B line
Branch prediction	24K bimodal / 24K gshare / 4K BTB entries, 32 RSB entries
L2 unified cache	2M, 8 ways, 64B lines, 5 cycles hit/miss indication
Memory latency	300 cycles
SOE event	L2 miss
SOE switch resource draining	6 cycles
SOE switch latency	Cycle accurately simulated. Usually takes around 25 cycles (depending on the instructions mix)
λ (cycles between $IPSw_j$ update)	250,000 cycles
Maximum cycles quota per thread	50,000 cycles

4.1 Machine Configuration

We simulated an out-of-order core with first-level instruction and data caches, a unified second-level cache (L2), a pipelined bus, and a constant latency memory (as shown in Figure 5). Table IV summarizes machine configuration parameters. Structure sizes were based on Intel’s disclosure of their latest processor core [Krewell 2006] and were slightly increased to reflect our view on a future version of that processor. Memory latency was set to 300 cycles, which is 75 ns at 4 GHz processor frequency. The simulated processor pipeline as well as the memory hierarchy are shown in Figure 5.

We have modified the simulator to support SOE multithreading, switching on L2 cache misses (last-level cache misses). Misses induced by load instructions as well as i/d TLB page walks are tracked by flagging them in the ROB (re-order buffer). A thread switch is triggered when the head of the ROB (the next

instruction or microoperation to be retired) is flagged as handling a miss, which has not been resolved.⁷ A thread switch causes draining of instructions from the RS (reservation station), ROB and LB (load buffers). Resource draining is simulated as a six-cycle long draining. Switch latency, the number of cycles from the start of the switch until the first instruction of the next thread retires, is cycle accurately simulated. The switch latency is not constant (depending, for example, on the instruction that was switched in), and usually accumulates to around 25 cycles.

Structures such as iTLB, dTLB, caches, and branch prediction history are shared, and are not flushed on switch. This is required in order to maintain performance after thread switch events [McNairy and Bhatia 2005]. Some structures, such as the 128 entries 4-way d-TLB, include a thread tag per entry in order to remove the flushing requirement. The L1 D-cache is physically tagged (does not require thread tag per entry). The store buffer keeps dispatching retired stores even after a flush, but will not forward their data if they are not from the same thread.

We set λ to 250,000 cycles. λ is the period used for sampling hardware counters and recalculating fairness parameters. In order to ensure that all threads are run in every λ cycles, each thread is limited to a certain amount of time before it is forced to switch out. We refer to this value as the maximum cycles quota per thread. This value should be less than λ/N , where N is the number of threads. A value less than λ/N ensures that all threads execute in any given λ cycles. The maximum cycles quota deals with the rare cases, where threads do not encounter any miss in λ cycles. It should be noted that the maximum cycles quota for a thread switch should be large enough so that the quota-forced switches are relatively rare and do not cause performance degradation. We used 50,000 cycles as the maximum cycles quota per thread.

Spec CPU2000 benchmarks [CPU2000] were simulated on a two-thread SOE configuration. Caches were warmed up using 10,000,000 instructions from each thread. Threads were simulated until both of them completed at least 6,000,000 instructions. The first 1,000,000 simulated instructions were not included in the results (statistics) and were used to warm up the internal microarchitecture state (internal structures, such as the branch prediction tables, as well as the fairness mechanism state). When the same benchmark was run on both threads, the two threads were offset by 1,000,000 instructions.

We used 31 combinations of benchmarks. In 13 cases, we used the same benchmark (at the same starting point) on both threads. Each combination was simulated using SOE without fairness ($\mathbb{F} = 0$), and with $\mathbb{F} = 1, \frac{1}{2},$ and $\frac{1}{4}$. In addition, we simulated each benchmark alone on the processor, in order to obtain its real IPC_j^{ST} . We use *name-number* as naming convention to refer to the executed benchmark application on each thread, where *name* is the application name and *number* is the point we used in that execution (*number* is an enumeration of starting points at regular intervals).

⁷This triggering scheme allows overlapping of clustered misses executed using the out-of-order (prefetching) effect.

Fig. 6. IPC_j^{ST} estimation using hardware counters while running two threads in SOE multithreading with fairness enforcement (top). The estimated speedup of the individual threads when run with and without fairness enforcement (middle). The achieved ratio between thread speedups (achieved fairness) with and without fairness enforcement (bottom). Fairness is enforced to $\frac{1}{4}$. Markers drawn every 250,000 cycles (in all three charts).

5. SIMULATION RESULTS

5.1 Detailed Examination

We use a representative two-thread combination in order to gain insight into the fairness-enforcement mechanism, as well as to how IPC_j^{ST} is estimated while running in SOE. We use a combination of *gcc-072* and *eon-015* applications. This combination requires active fairness enforcement without which the gcc thread almost starves, while eon runs on the processor most of the time.

Fairness enforcement is done by calculating $IPSw_j$ for each thread, based on the estimated values of IPC_j^{ST} . IPC_j^{ST} is estimated using hardware counters. Figure 6 (top) shows single-thread performance of the two threads when each of them is executed alone on the processor, compared to the estimated performance using hardware counters. Eq. (14) is used to estimate the performance. The figure shows that hardware counters can be effectively used to estimate single thread performance of each thread (IPC_j^{ST}), while they are both running in SOE.

As shown in Figure 6 (top), the estimated IPC_j^{ST} closely tracks the real IPC_j^{ST} . It is usually slightly lower than the real IPC_j^{ST} , because of several issues that may slow down each thread's execution in between misses. When a thread executes alone, the out-of-order mechanism uses some of the miss-stall cycles for out-of-order execution. These speculatively executed instructions are retired after the miss is resolved. Needless to say, these executions are not possible in SOE, which uses the stall time for the execution of the other thread. Another factor which reduces the performance of the individual threads is resource sharing (e.g., branch predictor tables). Sharing of resource reduces their effective size, as seen by each thread, when both threads are running together.

Figure 6 (middle) shows the individual thread estimated speedups and the actual achieved fairness (bottom), when the two threads run under SOE. When fairness is not enforced, the second thread (eon) executes most of the time, which causes the first thread (gcc) to almost starve. When fairness is enforced to $\frac{1}{4}$, the second thread's (eon) speedup is not allowed to exceed the first thread's (gcc) speedup by a factor of more than four. Now the speed of gcc is 20 times faster than its speed without fairness enforcement.

The speedup plot shows one occurrence in which the second thread gets higher speedup, followed by a slightly higher speedup of the first thread. This can be seen on the plots at 6,000,000 cycles when no fairness is enforced and on 7,000,000 cycles when fairness is enforced to $\frac{1}{4}$. This is most likely because of a short performance phase change, in which the estimation based on the previous λ (250,000) cycles is not accurate. This causes a short unfair execution. In long runs, however, the effects of these short unfair periods average toward an average of good fairness. The time difference of 1,000,000 cycles indicates that phase change occurred in the second thread (eon), which gets slower when fairness is enforced (had the phase change been in the other thread, which gets faster when fairness is enforced, the occurrence would have moved to an earlier cycle in the fair scenario).

5.2 Fairness and Throughput

The weighted speedup and aggregate IPC of different thread combinations are shown in Figure 7. The average weighted speedup is 1.12, 1.13, 1.14, and 1.14, for no fairness enforcement ($\mathbb{F} = 0$), $\mathbb{F} = \frac{1}{4}$, $\frac{1}{2}$, and 1. The average increase in aggregate IPC (over $(IPC_1^{ST} + IPC_2^{ST})/2$) is 26, 23, 21, and 16%. The aggregate IPC slightly drops when fairness is enforced because of the diversion toward the slower thread (which improves the weighted speedup).

In most cases, as shown in Figure 7a, weighted speedup improves as fairness strictness increases. In some cases, however, weighted speedup slightly drops when strict fairness of $\mathbb{F} = 1$ is enforced (e.g., *mgrid-019:art-003*, *mgrid-011:mgrid-011*). This slight drop is most probably the result of over strictness, without which fairness would have averaged over time (this is supported by the distribution of switches shown later in Figure 11). Assume, for example, that execution is slightly biased toward one thread in one-half of the time and to the other thread in the other one-half. Enforcing strict fairness, in this case,

(a) SOE weighted speedup $\left(WS = \frac{IPC_1^{SOE}}{IPC_1^{ST}} + \frac{IPC_2^{SOE}}{IPC_2^{ST}} \right)$.

(b) SOE aggregate IPC ($IPC^{SOE} = IPC_1^{SOE} + IPC_2^{SOE}$) with and without fairness enforcement, as well as IPC_j^{ST} of each of the two threads when executed alone.

Fig. 7. SOE weighted speedup and aggregate IPC.

would induce thread switches to remove the biasing in each one-half. Not all of these induced switches, however, were required over the whole run, in which the biasing would have averaged anyway. Inaccuracies in performance estimation (e.g., execution phase change) can also cause weighted speedup drop in the case of strict fairness ($\mathbb{F} = 1$).

Usually, as shown in Figure 7b, fairness enforcement has only negligible effect on the aggregate IPC when IPC_j^{ST} of the two threads is roughly the same (e.g., *lucas-029:applu-011*, *bzip2-048:bzip2-048*). The greater the difference between performance of the individual threads, the more thread switches will be required in order to enforce fairness. Induced switches divert execution toward one of the threads, which can cause decrease in aggregate IPC as enforced fairness $\mathbb{F}$ increases (e.g., *galgel-013:gcc-065*, *apsi-022:swim-015*) or an increase in aggregate IPC (e.g., *facerec-004:wupwise-005*). An increase in aggregate IPC

Fig. 8. SOE weighted speedup and switches induced by fairness enforcement.

is the result of fairness enforcement diverting execution toward the “faster” thread (the thread with the higher IPC_{no_miss}). It should be noted that a decrease in aggregate IPC does not necessarily mean lower weighted speedup (as shown in Figure 7a).

Figure 8 shows weighted speedup change caused by fairness enforcement, as well as the number of forced thread switches in 1000 cycles. Forced switches are caused by fairness enforcement. Enforcing fairness of 1, $\frac{1}{2}$ or $\frac{1}{4}$ causes an average weighted speedup improvement of 2.0, 1.6 and 0.5% respectively. As shown, weighted speedup is not affected by fairness enforcement when there is small number of induced thread switches (middle of the chart). In these cases, fairness enforcement has minimal effect on the execution. Fairness enforcement, however, can cause an increase (right side of the chart) or even a decrease in weighted speedup (left side of the chart). Only when strict fairness ($\mathbb{F} = 1$) is enforced does the decrease of weighted speedup seem to have some significance. As already explained, this is most probably the result of fairness issues that average over time even without strict fairness enforcement and inaccuracies of performance estimation. Using fairness of $\frac{1}{2}$ or $\frac{1}{4}$, allows the thread speedups to have some difference (slack), which reduces weighted speedup change, and almost eliminates the weighted speedup decrease (left side of the chart). It should be noted that since an average thread switch overhead is 25 cycles, the overhead of the additional thread switches is small, and, on average, is 2.1, 0.9 and 0.4% for $\mathbb{F} = 1$, $\frac{1}{2}$ and $\frac{1}{4}$, respectively.

Figure 9 shows achieved fairness with and without fairness enforcement. The $\mathbb{F} = 0$ line shows the achieved fairness of the two threads without any attempt to enforce it. The lines with $\mathbb{F} = \frac{1}{4}$, $\frac{1}{2}$, and 1 show the fairness when it is enforced to $\frac{1}{4}$, $\frac{1}{2}$, and 1 respectively. Simulation runs are ordered by their achieved fairness when no fairness is enforced. In almost all cases, as shown in the figure, fairness enforcement achieves a fairness close to $\mathbb{F}$ (the target fairness being enforced). In a few cases, when strict fairness is enforced ($\mathbb{F} = 1$), achieved fairness is

Fig. 9. SOE achieved fairness for $\mathbb{F} = 0$ (no fairness enforcement), $\frac{1}{4}$, $\frac{1}{2}$, and 1.

Fig. 10. SOE average and standard deviation of $\min(\mathbb{F}, \text{Achieved}_{\text{fairness}})$.

0.8 or less. This is probably because of inaccuracies in performance estimation and tracking, largely as a result of the additional induced thread switches. Using less strict fairness eliminates these cases completely. On runs, which are also fair without fairness enforcement, the mechanism has small effect on the achieved fairness.

Figure 10 shows the average achieved fairness. It shows the average and standard deviation of $\min(\mathbb{F}, \text{Achieved}_{\text{fairness}})$. Using $\min(\mathbb{F}, \text{Achieved}_{\text{fairness}})$ in our calculation ensures that when achieved fairness is greater than $\mathbb{F}$ (the target

Fig. 11. Distribution of thread switches in absolute values (top) and percentage (bottom). Forced switches are caused by fairness enforcement.

fairness being enforced), it will be truncated to $\mathbb{F}$. This gets rid of the biasing effect of runs that are fair even without fairness enforcement (these runs that would have otherwise biased the averages toward 1). There is no truncation for the case of $\mathbb{F} = 0$. For $\mathbb{F} = \frac{1}{4}$ the standard deviation is small indicating that the achieved fairness is highly accurate. The standard deviation increases as stricter fairness is being enforced.

5.3 Thread Switches

Figure 11 shows the distribution of thread-switches for 15 of our simulations. It shows the number of switches per 1,000,000 cycles categorized by their cause:

1. Maximum cycles quota—switches caused by a thread running for a long time without switching out (e.g., not encountering a miss event).
2. L2 miss—switches triggered by a retirement stall resulting from a nonresolved L2 miss.
3. Forced switches—switches caused by fairness enforcement.

As shown in Figure 11, when fairness is not enforced ($\mathbb{F} = 0$), most switches are because of L2 misses. In some thread combinations, which have low L2

Fig. 12. Forced thread switches overhead (percentage of cycles wasted because of the additional thread switches).

miss rates (misses per cycle), some of the switches are caused by the maximum cycles quota. Threads that have switches caused by maximum cycles quota usually have a low absolute number of switches. Enforcing fairness causes forced switches, as expected. The number of forced switches increases as enforced fairness becomes more strict.

It should be noted that the maximum cycles quota causes thread switches in workloads that have a low number of L2 misses (max cycles quota is a fixed constant quota of cycles per switch, different from $IPSw_j$, which is fairness enforcement instructions quota). The number of thread switches triggered by this quota decreases dramatically when fairness is enforced. That is because when one of the threads runs for a long time it will be switched out because of fairness enforcement before its max cycles quota is reached. The max cycles quota is extremely important when no fairness is being enforced. It is also required when fairness is being enforced in order to maintain functionality for the rare extreme cases of low miss rate in both threads. Switches triggered by maximum cycles quota decrease performance since they induce additional switching overheads (thread switch latency). This performance degradation, however, is negligible, as can be seen by the low absolute number of such thread switches. Needless to say, setting maximum cycles quota to a low value will cause performance loss. Maximum cycles quota should be less than $\frac{\lambda}{N}$, where N is the number of threads. This will guarantee that all threads get to run in each λ cycles. We used $\frac{\lambda}{2.5N}$ in our simulations, which is still large enough to cause negligible performance loss.

Figure 12 shows the overhead of the additional thread switches. It shows the percentage of cycles that are used for the pipeline flushes caused by the fairness enforcement. As shown, the overhead is relatively low. This means that aggregate IPC drop, which is usually higher than the additional thread switches overhead, is mainly caused by the diversion of the processor execution from a high IPC thread to a thread with low IPC.

Fig. 13. Sensitivity analysis of achieved fairness (left) and weighted speedup (right) to λ value.

5.4 Sensitivity to λ Values

Hardware counters are sampled every λ cycles in order to calculate fairness parameters. These parameters are then used in the following λ cycles for controlling fairness. λ is an important parameter of fairness mechanism implementation. Reducing λ reduces the statistical significance of the sampled counters, while increasing its value slows down the mechanism's ability to track performance phases of the workloads.

Figure 13 shows sensitivity analysis of fairness and weighted speedup to λ value. We simulated λ values of 25,000 to 500,000 cycles, with the max cycles quota set to $\frac{\lambda}{5}$. Figure 13a shows the average fairness achieved⁸ for the different λ values. The average fairness is high even when enforced to $\frac{1}{4}$ or $\frac{1}{2}$ because the average is pulled upward by simulations that achieve high fairness even without enforcement. The figure shows that fairness is stable over λ range of 25,000 to 500,000 cycles. There is a small drop in achieved fairness for small and large λ values (50,000 and 250,000 cycles). This drop is up to 4.5% (compared to the highest achieved fairness), which is quite negligible.

Figure 13b shows the average of the weighted speedup for the different λ values, normalized to weighted speedup when $\lambda = 250,000$. It shows that the highest weighted speedup is achieved at $\lambda = 250,000$. Weighted speedup drops by up to 1.5% for small and large λ values. This shows that, like fairness, weighted speedup is not sensitive to λ over the range of 25,000 to 500,000, with a small drop for small and large λ values.

5.5 Sensitivity to Memory Latency

Our SOE implementation hides memory latencies caused by L2 misses. It is, therefore, important to investigate the effect of memory latency on the fairness enforcement mechanism. Figure 14 shows the effect of different memory latencies on SOE weighted speedup and achieved fairness.

Figure 14a shows SOE weighted speedup. As expected, SOE weighted speedup increases with memory latency. This is because the performance of SOE is less sensitive to memory latency than that of a single threaded run.

⁸Note the difference between Figures 10 and 13a. Figure 10 shows the average of $\min(F, \text{Achieved}_{\text{fairness}})$; Figure 13a displays the achieved fairness.

Fig. 14. Sensitivity analysis of weighted speedup (top) and fairness (bottom) to memory latency.

SOE performance only slightly degrades, when memory latency is increased, as a result of the increase in overlapping between misses of different threads (time in which both threads are handling misses and no useful execution can be done). Single-threaded performance, however, will significantly drop as execution stalls on L2 misses, waiting for values to be returned from the memory.

Figure 14b shows the average achieved fairness for different memory latencies. It shows that achieved fairness changes by no more than 5% for any $\mathbb{F}$, for memory latencies of 180 to 600 cycles. This means that the fairness mechanism operates well over this large range of memory latencies.

Figure 14c shows average achieved fairness normalized to its value at 180 cycles memory latency. Normalizing the achieved fairness makes it easier to view the trend of the achieved fairness. When no fairness is enforced ($\mathbb{F} = 0$) archived fairness improves as memory latency increases. This can be seen in our definition of fairness (5) $Fairness \equiv \min_{j,k} (\frac{CPM_j + Miss_{latency}}{CPM_k + Miss_{latency}})$: as memory latency ($Miss_{latency}$) increases, it becomes more dominant in the calculation.

Figure 14c shows that the tendency of achieved fairness to grow with memory latency is reduced as stricter fairness is being enforced. When $\mathbb{F} = 1$ (perfect fairness enforcement), the tendency is reversed and achieved fairness decreases as memory latency grows. This is most probably the effect of our model

Fig. 15. Analysis of our simulation runs using the harmonic mean of the individual threads' speedups.

assuming that a miss is resolved by the time its thread regains execution.⁹ This, however, is not always the case, as some misses are clustered and may cause short execution of threads (before they relinquish execution and the other thread regains it). Naturally, the probability of such occurrences increases with memory latency. This causes inaccuracy in the calculation of fairness parameters and reduces the achieved fairness. The effect of this inaccuracy is more dominant when stricter fairness is enforced. This decreases the slope of the achieved fairness curve. Clustered misses also have some effect on our fairness calculation when no fairness is being enforced, but it is of second order.

5.6 Harmonic Mean of Speedup

Luo et al. [2001] used the harmonic mean of the individual threads' speedups in order to combine both fairness and throughput into a single metric.

We found the harmonic mean to give insufficient insight into either throughput or fairness. We prefer the use of two metrics, one for fairness and one for throughput improvement (weighted speedup). Each of them clearly captures its purpose. Using separate metrics allows the user to subjectively balance between fairness and throughput. For completeness, we evaluate our approach using the harmonic mean of Luo et al. [2001].

Figure 15 shows our simulation results analyzed using the harmonic mean of the individual threads' speedups. The figure shows the harmonic mean metric value for when no fairness is enforced ($F = 0$) and when it is enforced to 1,

⁹This is assumed in Eq. (2).

$\frac{1}{2}$ and $\frac{1}{4}$. As shown, our mechanism improves (increases) the harmonic mean value in cases when harmonic mean is low without fairness enforcement (left side of the figure) and has small effect when harmonic mean is high without fairness enforcement (right side of the chart). Our fairness enforcement works well according to the harmonic mean metric.

6. DISCUSSION

Simple heuristics for fairness improvements, such as fetch prioritization or simple time-sharing, are ineffective for SOE. Fetch prioritization [Luo et al. 2001; Sun et al. 2005] is a fine-grained prioritization approach appropriate for SMT. It prioritizes fetches from the different threads. This can result in changing the thread from which instructions are fetched every few cycles. This is applicable to SMT, where the front end maintains several threads and fetches are based on prioritization. SOE, however, maintains a single fetch thread and has to flush the pipeline on each such thread switch. Fetch prioritization will result in frequent pipeline stalls which will severely impair SOE performance.

Simple time sharing is not useful for enforcing fairness, as shown in Example 3, because of its inability to adjust to each workload and execution phase. In some cases (thread combinations), it may result in high throughput, while in others it may achieve low fairness and throughput. It is not possible to tune simple time-sharing scheme to maintain either throughput or fairness, as its performance (fairness and throughput) depends on the actual threads mix.

The empirical results shown in this paper demonstrate that fairness can be effectively maintained using the proposed mechanism. The results also indicate that attempting to maintain perfect fairness ($\mathbb{F} = 1$) has, in some cases, negative effects on performance (weighted speedup) and, in some cases, on the achieved fairness as well. Both weighted speedup and the achieved fairness are affected by the large number of forced switches introduced by the attempt to maintain perfect fairness. These thread switches reduce the accuracy of the IPC^{ST} estimation, on which fairness enforcement mechanism is based. In extreme cases, this can lead to achieving worse fairness than when attempting to enforce a less strict one (e.g., *gcc-065:gcc-065* or *mgrid-011:mgrid-011* in Figure 9).

Increasing the enforced fairness ($\mathbb{F}$) increases the number of induced thread switches. In some cases this can have negative effect on aggregate IPC and weighted speedup. Aggregate IPC can drop when fairness enforcement is too strict, inducing thread switches that are not really required (because of averaging effect or to inaccuracies in performance estimation as a result of execution phase changes). Simply increasing λ will reduce the accuracy in which the mechanism track phases in the programs and will decrease the throughput and achieved fairness. Using strict fairness introduces cases where fairness mechanism cause weighted speedup decrease. Hence, there is a balance between fairness and weighted speedup. The balance point between fairness and weighted speedup depends on the user's preferences. It is extremely hard to produce an objective metric that combines both throughput and fairness to allow such balance point analysis. According to our simulation results, using $\mathbb{F} = \frac{1}{2}$

is a reasonable compromise that almost completely eliminates cases in which the weighted speedup decreases.

It should be noted that the increase of thread switches caused by fairness enforcement has a negative effect on throughput, as those pipeline flushes waste execution cycles. However, as long as the overhead of the additional thread switches is less than the latency of an external memory access, we still gain more than we lose. In our case, we gain execution cycles as long as fairness enforcement induces less than 12 thread switches per memory miss (each thread switch costs around 25 cycles and an external memory access is 300 cycles long). In other words, there should not be more than 12 switches before any thread encounters a miss. This is extremely rare in a two-thread model and can happen in a rare cases of phase changes when performance is inaccurately tracked by our mechanism.¹⁰

All empirical results support the validity of our analytical model. This model provides a good explanation for our empirical results, such as the effect of fairness enforcement on throughput as well as that of memory latency on achieved fairness. Our results show that assumptions made to simplify the model¹¹ have a small effect on its validity.

The SOE mechanism presented in this paper may be expanded by using various events to trigger thread switches. For example, L1 misses (which may hit or miss the L2 cache) can cause a thread switch to hide L1 miss latency. IO events can be used to trigger switches to hide long IO operations, such as disk IO. Another example are explicit instructions that can also trigger thread switches.¹²

We used a constant predefined value for miss latency (300 cycles) to represent the average memory access latency. Some switch events may have variable latencies, whose average is hard to predict (e.g., L1 miss). In these cases, the event's latency should be monitored using hardware counters. For example, in order to determine L1 miss latency, a hardware counter could count the total number of cycles used for L1 misses handling. On every λ cycles, when fairness parameters are recalculated, $Miss_{latency}$ should also be calculated, using the hardware counter divided by the number of L1 misses. This method can be used to support multiple event types with variable latencies.

The analytical model and the fairness enforcement mechanism can be extended to also support thread priorities. In this case, threads of different priorities will be able to share the processor while fairness enforcement will take their priorities into consideration. We leave this extension for future work.

¹⁰Let us examine an extreme scenario to illustrate the effects of inaccurate performance tracking. Assume, for example, that one thread has a high miss rate, which causes both threads to have a low $IPSw$ (see Eq. 9). Now assume that as soon as $IPSw$ is calculated, execution phase changes to one in which both threads have low miss rates. In this case, there may be many consecutive forced thread switches before a single miss is encountered.

¹¹An example of such an assumption is that any miss is resolved by the time its thread regains execution.

¹²An example of such instruction in X86 instruction set is *pause*. This instruction hints that a short execution pause can be done. *Pause* is usually used in busy wait loops that wait for external events.

7. CONCLUSIONS

SOE is a low-power, low-complexity multithreading solution, that improves processor utilization and throughput. It hides execution stalls, such as last-level cache misses, by switching threads on the detection of such stalls. This paper presented a fairness enforcement mechanism for SOE, forcing thread switch points based on architectural level runtime fairness tracking. Fairness tracking is done by estimating the single-thread performance of individual threads, while they are running using SOE multithreading, based on a model developed in this paper. The overhead of the fairness mechanism is very low. It requires the use of a few hardware counters, which exist anyhow in most modern processors, and the addition of hardware or software hooks (e.g., interrupts) for instructions-per-switch ($IPSw_j$) calculation.

Simulation results show that the suggested fairness enforcement mechanism works well for a variety of applications. SOE achieves an average aggregate IPC improvement (over single thread) of 26, 23, 21, and 16% for $\mathbb{F} = 0$ (no fairness enforcement), $\frac{1}{4}$, $\frac{1}{2}$, and 1, respectively. As shown, the average aggregate IPC drop when fairness is enforced is due to the diversion of execution toward a lower IPC thread, and does not reflect the overhead of the additional induced thread switches.

SOE achieves an average weighted speedup over single thread of 1.12 when no fairness is enforced. In this case, a sixth of our runs achieved poor fairness in which one thread ran extremely slowly (10 to 100 times slower than its single-thread performance), while the other thread's performance was hardly affected. By using the proposed mechanism, we guarantee fairness of $\frac{1}{4}$, $\frac{1}{2}$, and 1 while improving the weighted speedup by 0.5, 1.6, and 2.0%, respectively. The average weighted speedup increases, because fairness enforcement usually slightly reduces the speedup of one thread and significantly increases the speedup of the other.

When thread execution is unfair, the fairness mechanism improves fairness by forcing additional thread switch points. Fairness enforcement, when not required (threads run in a fair fashion even without any enforcement), has a negligible effect on the execution. In some cases, inaccuracies in performance estimation (and tracking) can cause a light-weighted speedup drop when strict fairness is enforced. We believe that enforcing strict fairness is not necessary. A reasonable compromise is to enforce a fairness of $\frac{1}{2}$ in order to reduce the impact of thread switches and possible inaccuracies.

Our simulations show that the proposed fairness enforcement mechanism performs well for a large range of memory latencies. Results also show that fairness enforcement is effective for a large range of λ values (the period at which fairness enforcement parameters are recalculated), with best performance achieved for the range of 50,000 to 250,000 cycles.

The fairness mechanism is based on estimating the single-thread performance of the individual threads, while running in SOE. This estimated performance is used to calculate the achieved fairness, and to induce thread switches in case the achieved fairness needs to be improved. The extension of our method to SMT is by no means a trivial task because of the difficulty in estimating

the performance of the individual threads while they are running in SMT. Extension of this work to SMT is being considered and remains as a future work.

REFERENCES

- AGARWAL, A., LIM, B.-H., KRANZ, D., AND KUBIATOWICZ, J. 1990. April: A processor architecture for multiprocessing. *SIGARCH Comput. Archit. News* 18, 3a, 104–114.
- AGARWAL, A., BIANCHINI, R., CHAIKEN, D., JOHNSON, K. L., KRANZ, D., KUBIATOWICZ, J., LIM, B.-H., MACKENZIE, K., AND YEUNG, D. 1995. The MIT Alewife machine: architecture and performance. In *Proc. of ISCA-22*. 2–13.
- AKKARY, H., SRINIVASAN, S. T., KOLTUR, R., PATIL, Y., AND REFAAI, W. 2004. Perceptron-based branch confidence estimation. In *Proc. of HPCA-10*. 265.
- BALAKRISHNAN, S., RAJWAR, R., UPTON, M., AND LAI, K. 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proc. of ISCA-32*. 506–517.
- BAZEGHI, C., MESA-MARTINEZ, F. J., AND RENAU, J. 2005. uComplexity: Estimating processor design effort. In *Proc. of MICRO-38*. 209–218.
- BORKENHAGEN, J. M., EICKEMEYER, R. J., KALLA, R. N., AND KUNKEL, S. R. 2000. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development* 44, 6, 885–898.
- CAI, G. 2000. Power-sensitive multithreaded architecture. In *Proc. of the 2000 IEEE International Conference on Computer Design (ICCD '00)*. 199.
- CAZORLA, F. J., KNIJNENBURG, P. M., SAKELLARIOU, R., FERNANDEZ, E., RAMIREZ, A., AND VALERO, M. 2004a. Feasibility of QoS for SMT by resource allocation. In *Intl. EuroPar Conference*. 535–540.
- CAZORLA, F. J., RAMIREZ, A., VALERO, M., AND FERNANDEZ, E. 2004b. Dynamically controlled resource allocation in SMT processors. In *Proc. of MICRO-37*. 171–182.
- COOKSEY, R., JOURDAN, S., AND GRUNWALD, D. 2002. A stateless, content-directed data prefetching mechanism. *SIGOPS Oper. Syst. Rev.* 36, 5, 279–290.
- CPU2000. Standard performance evaluation corporation, Spec CPU2000.
- DAVIS, J. D., LAUDON, J., AND OLUKOTUN, K. 2005. Maximizing CMP throughput with mediocre cores. In *Proc. of PACT-14*. 51–62.
- DUDA, K. J. AND CHERITON, D. R. 1999. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*. 261–276.
- EICKEMEYER, R. J., JOHNSON, R. E., KUNKEL, S. R., LIM, B.-H., SQUILLANTE, M. S., AND WU, C.-F. E. 1997. Evaluation of multithreaded processors and thread-switch policies. In *Proc. of the International Symposium on High Performance Computing (ISHPC '97)*. 75–90.
- FALCON, A., STARK, J., RAMIREZ, A., LAI, K., AND VALERO, M. 2004. Prophet/critic hybrid branch prediction. *SIGARCH Comput. Archit. News* 32, 2, 250.
- FARRENS, M. K. AND PLESZKUN, A. R. 1991. Strategies for achieving improved processor throughput. In *Proc. of ISCA-18*. 362–369.
- GLASKOWSKY, P. N. 2003. IBM previews Power5. *Microprocessor Report*.
- GOCHMAN, S., MENDELSON, A., NAVEH, A., AND ROTEM, E. 2006. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal*.
- GRUENEWALD, W. AND UNGERER, T. 1997. A multithreaded processor designed for distributed shared memory systems. In *Proc. of the 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*. 206.
- GUPTA, A., HENNESSY, J., GHARACHORLOO, K., MOWRY, T., AND WEBER, W.-D. 1991. Comparative evaluation of latency reducing and tolerating techniques. *SIGARCH Comput. Archit. News* 19, 3, 254–263.
- GWENNAP, L. 1995. Intel's P6 uses decoupled superscalar design. *Microprocessor Report* 9, 2 (Feb.).
- HASKINS, J. W. AND SKADRON, J. 2001. Inexpensive throughput enhancement in small-scale embedded microprocessors with block multithreading: Extensions characterization, and tradeoffs. In *Proc. of the 20th IEEE International Performance, Computing, and Communications Conference*. 319–328.

- JONES, M. B., ROŞU, D., AND ROŞU, M.-C. 1997. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. *ACM SIGOPS Operating Systems Review* 31, 5, 198–211.
- KAHLE, J. 2005. The cell processor architecture. In *Proc. of MICRO-38*. 3.
- KALLA, R., SINHARROY, B., AND TENDLER, J. 2004. IBM Power5 chip: A dual-core multithreaded processor. In *Proc. of MICRO-37*. 40–47.
- KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip multi-processor architecture. In *Proc. of PACT-13*. 111–122.
- KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. 2005. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro* 25, 2, 21–29.
- KREWELL, K. 2004. AMD vs. Intel in dual-core duel. *Microprocessor Report*.
- KREWELL, K. 2006. Intel looks to core for success. *Microprocessor Report*.
- KUMAR, R., TULLSEN, D. M., JOUPPI, N. P., AND RANGANATHAN, P. 2005. Heterogeneous chip multi-processors. *Computer* 38, 11, 32–38.
- LUO, K., GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In *Proc. of the International Symposium on Performance Analysis of Systems and Software*. 164–171.
- MARR, D., BINNS, F., HILL, D., HINTON, G., KOUFATY, D., MILLER, J., AND UPTON, M. 2002. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal* 6.
- McNAIRY, C. AND BHATIA, R. 2005. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro* 25, 2, 10–20.
- MOWRY, T. AND RAMKISSOON, S. 2000. Software-controlled multithreading using informing memory operations. In *Proc. of the 6th International Symposium on High-Performance Computer Architecture (HPCA '00)*. 121–132.
- MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proc. of HPCA-9*. 129.
- RAASCH, S. E. AND REINHARDT, S. K. 1999. Applications of thread prioritization in SMT processors. In *Proc. of the Workshop on Multithreaded Execution And Compilation*.
- RAASCH, S. E. AND REINHARDT, S. K. 2003. The impact of resource partitioning on SMT processors. In *Proc. of PACT-12*. 15.
- RATTNER, J. 2005. Multi-core to the masses. In *Proc. of PACT-14*. 3.
- SAZEIDES, Y. AND JUAN, T. 2001. How to compare the performance of two SMT microarchitectures. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, 180–183.
- SHREEDHAR, M. AND VARGHESE, G. 1996. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.* 4, 3, 375–385.
- SINGHAL, R., VENKATRAMAN, K., COHN, E., HOLM, J., KOUFATY, D., LIN, M.-J., MADHAV, M., MATTWANDEL, M., NIDHI, N., PEARCE, J., AND SESHADRI, M. 2004. Performance analysis and validation of the Intel Pentium4 processor on 90nm technology. *Intel Technology Journal* 8.
- SNARELY, A. AND TULLSEN, D. M. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. *SIGPLAN Not.* 35, 11, 234–244.
- SPRACKLEN, L. AND ABRAHAM, S. G. 2005. Chip multithreading: opportunities and challenges. In *Proc. of HPCA-11*. 248–252.
- SUN, C., TANG, H., AND ZHANG, M. 2005. An instruction fetch policy handling l2 cache misses in smt processors. In *Proc. of the 8th International Conference on High-Performance Computing in Asia-Pacific Region*. 519–525.
- TENDLER, J. M., DODSON, J. S., FIELDS, J. S., JR., H. L., AND SINHARROY, B. 2002. POWER4 system microarchitecture. *IBM Journal of Research and Development* 46, 1, 5–26.
- THEKKATH, R. AND EGGERS, S. J. 1994. The effectiveness of multiple hardware contexts. *SIGPLAN Not.* 29, 11, 328–337.
- TULLSEN, D. M. AND BROWN, J. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *Proc. of MICRO-34*. 318–327.
- TULLSEN, D. M., EGGERS, S., EMER, J., LEVY, H., LO, J., AND STAMM, R. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of ISCA-23*. 191–202.

- TULLSEN, D. M., EGGERS, S., AND LEVY, H. 1998. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*. 533–544.
- TUNE, E., KUMAR, R., TULLSEN, D. M., AND CALDER, B. 2004. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proc. of MICRO-37*. 183–194.
- UNGERER, T., ROBIC, B., AND SILC, J. 2003. A survey of processors with explicit multithreading. *ACM Comput. Surv.* 35, 1, 29–63.

Received August 2006; revised December 2006; accepted January 2007