Fairness and Throughput in Switch on Event Multithreading

Ron Gabor[1,2] Shlomo Weiss[1] Avi Mendelson[2]
[1]Tel Aviv University, Israel, {rongabor@post, weiss@eng}.tau.ac.il
[2]Intel, Israel, {ron.gabor, avi.mendelson}@intel.com

Abstract

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 throughput. We present simulation results of the performance of Switch on Event multithreading. Switch on Event multithreading achieves an average speedup over single thread of 24% *when no fairness is enforced. In this case, over a third 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 can guarantee fairness of* $1/4$*,* $1/2$ *and* 1 *for a small performance loss of* 2.2%*,* 3.7% *and* 7.2% *respectively.*

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 Multi-Processor (CMP) - multiple processors (on die) that share some of the memory hierarchy; e.g. IBM's Power4 [42] and Intel Duo [16].
- Simultaneous Multi-Threading (SMT) in which instructions from multiple threads are fetched, executed and retired on each cycle, sharing most of the resources in the core [45, 46]; e.g. IBM's Power5 [23] and Intel's P4 [30].
- 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 [13, 15, 43]; e.g. IBM's RS64 IV [6] and Intel's Montecito[31].

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 [5], but it usually also means lower power.

There is an ongoing trend towards lower power and complexity of microprocessors. All major microprocessor vendors are going to chip multiprocessors with simple cores rather than complex superscalars [22, 23, 25, 26, 36]. In order to improve throughput and power efficiency, more cores and threads are squeezed into a single processor die [7, 11, 40]. Asymmetric cores integration can further improve power to performance ratio by integrating simple cores with larger, more complex ones [4, 28]. SOE is extremely important to this 'simple-cores' trend, as it can increase number of threads at a relatively small power, area and complexity costs. Given SOE's importance, we feel it deserves further research and study.

In all the current multi-threaded architectures fairness is a major problem. Lack of fairness is usually caused when resources are unfairly shared between 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 the fairness among most of the resources of the machine. Handling fairness in CMP is limited to memory access while handling it in SMT is micro-

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00

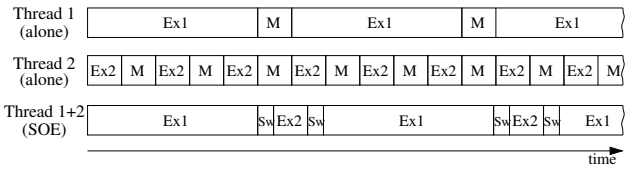

Figure 1. Intuitive example of unfair execution in SOE. ***Ex1*** **marks execution of instructions from thread 1,** ***Ex2*** **from thread 2,** ***M*** **marks last level cache misses and** ***Sw*** **denotes thread switch overheads. When both threads run together using SOE (bottom), the 2nd thread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

architectural dependent. Fairness in SOE, as will be shown in this paper, can be handled at the architectural level, freeing the microarchitecture from having to deal with it.

Example 1

The following example demonstrates the fairness problem in SOE. Consider the simple two threads 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.

1.1. 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 [15] 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.*[18] 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.*[13] studied SOE throughput in server workloads, and showed that SOE achieves its maximum throughput using three threads. Haskins *et al.*[20] 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 [47]. 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 [6] and Intel's Montecito[31]. Montecito preferred SOE over SMT due to 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 [1, 2, 17, 32], 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.*[48] explicit multithreading survey.

Fair cache sharing between multiple co-scheduled 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 [8, 24].

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 [12, 21]. We deal with the applicability of time sharing to SOE in Section 6.

Fairness of threads execution was studied in the context of SMT [29, 34, 35, 44]. Raasch and Reinhardt [35] showed that resource partitioning in SMT improves threads' execution fairness. For example, statically partitioned ROB improves fairness compared to competitive sharing. Luo *et al.*[29] 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). Several

attempts were made to suggest a single metric that combines both fairness and throughput. Snavely *et al.*[39] used weighted speedup in order to measure the goodness of their operating system level job scheduling. Weighted speedup is the sum of the individual threads' speedups[1]. This metric considers each instruction executed by a low IPC thread as being worth more. It is useful for increasing throughput in a fair manner when selecting a small number of jobs from a larger pool (OS tasks scheduling). Luo *et al.*[29] 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 further discuss these metrics in Section 6.

1.2. 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 individual threads' IPC, 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 tradeoffs. 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 run-time 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.

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 run-time.

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

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 (excluding miss latency).
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).
$\mathbb{F}$	Fairness to be enforced ($0 \leq \mathbb{F} \leq 1$).
λ	Period of fairness enforcement parameters calculation (in cycles).

Table 1. Abbreviations and symbols.

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).

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

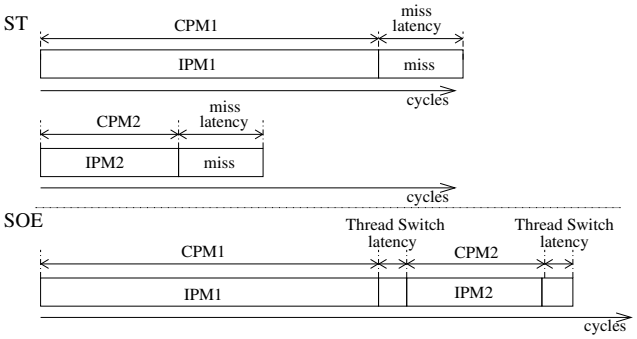

Figure 2. Representation of single thread runs for two threads (top) and SOE run of the same two threads (bottom).

an execution stall of $Miss_{lat}$ cycles ($Miss_{lat}$ 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 Equation 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 $CPM_j + Miss_{lat}$.

$$IPC_j^{ST} = \frac{IPM_j}{CPM_j + Miss_{lat}} \tag{1}$$

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 Equation 2, IPC_j^{SOE} can be calculated by dividing IPM_j by the average total 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 $Switch_{lat}$ cycles (the average overhead per thread switch).

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

It should be noted that Equation 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.

[2]We 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. It should be noted that our empirical results indicate that the analytical model gives adequate approximation. It should also be noted that our implementation only counts the first miss in a each group of overlapped misses (see Section 3.1).

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 [29]. We define a perfectly fair system as:

$$\forall j,k \; \frac{IPC_j^{SOE}}{IPC_j^{ST}} = \frac{IPC_k^{SOE}}{IPC_k^{ST}} \tag{3}$$

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

$$\begin{aligned} Fairness &\equiv \min_{j,k} \left(\frac{\left(\frac{IPC_j^{SOE}}{IPC_j^{ST}}\right)}{\left(\frac{IPC_k^{SOE}}{IPC_k^{ST}}\right)} \right) \\ &= \min_{j,k} \left(\frac{IPC_j^{SOE} \cdot IPC_k^{ST}}{IPC_j^{ST} \cdot IPC_k^{SOE}} \right) \end{aligned} \tag{4}$$

It should be noted that this definition of fairness is more strict than the harmonic mean used by Luo *et al.*[29]. It uses the maximum and minimum speedups for fairness calculation, which guarantees that these values will not have reduced impact on fairness due to 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 Equation 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 Equation 1 and Equation 2 into Equation 4 results in Equation 5:

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

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 due to last level cache misses then $IPSw_j = IPM_j$ and $CPSw_j = CPM_j$). Using $IPSw_j$ and $CPSw_j$ modifies Equation 2:

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

Substituting Equation 6 and Equation 1 into equation 4 we get:

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

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 Equation 7, a system achieves the required fairness $\mathbb{F}$ if it satisfies Equation 8.

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

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 Equation 9 is guaranteed to satisfy Equation 8[3].

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

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 due to $IPSw_j$. Hence, there is no way to increase $IPSw_j$ to a value greater than IPM_j (that is why we use $\min$ in Equation 9).

As shown in Equation 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 $\mathbb{F} = 1/2$ will guarantee a maximum speedup ratio of 2. This means that the ratio of speedups 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 throughput as the IPC^{SOE}, which is the IPC of all of the threads when executed together using SOE. As shown in Equation 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, Equation 6 can be used to show that $IPC^{SOE} = \sum_k IPC_k^{SOE}$:

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

[3]This can be proven algebraically.

	$\mathbb{F}=0$		$\mathbb{F}=1$		$\mathbb{F}=1/2$	
j (thread num)	1	2	1	2	1	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	

Table 2. Example of two threads SOE with and without fairness enforcement.

Example 2

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), the second every 1,000 instructions (400 cycles). Table 2 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 ($IPM_1 = 15,000$).

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

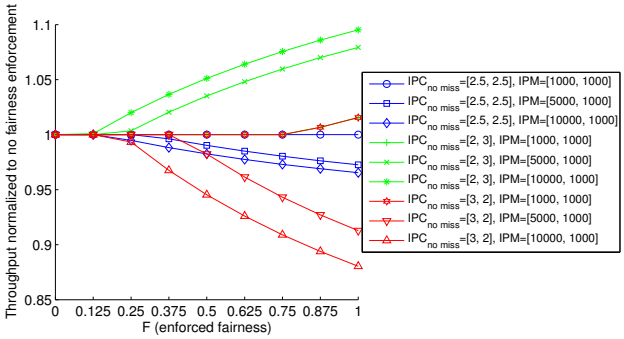

Figure 3. The effect of fairness enforcement on throughput, using a two threads analytical model. Legend's notation: $IPC_{no\_miss} = [IPC_{no\_miss1}, IPC_{no\_miss2}], IPM = [IPM_1, IPM_2]$**.**

2.5. Fairness and Throughput Analysis

When fairness is not enforced, every thread switch hides a memory access stall (last-level cache miss). However, when fairness is enforced, additional thread switches are induced in order to maintain the required fairness. Intuitively, these forced thread switches, which cause a $Switch_{lat}$ cycles stall, reduce the throughput of SOE[4].

Figure 3 shows the effect of fairness enforcement on throughput in a two threads model. It shows throughput degradation due to 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 cause throughput degradation. As shown by the lines of $IPC_{no\_miss} = [2.5, 2.5]$, when IPC_{no_miss} is similar for both threads, throughput degrades by up to 4%. However, when the two threads do not have the same IPC_{no_miss}, throughput can degrade by up to 15% or can actually improve by up to 10%. The throughput increase in the $IPC_{no\_miss} = [2, 3]$ cases is explained by noting that fairness enforcement biases the execution towards the thread with the higher IPC_{no_miss}, hence, improves throughput.

Figure 3 shows a tradeoff between $\mathbb{F}$ (enforced fairness) and throughput (for $Miss_{lat} = 300$, $Switch_{lat} = 25$). As shown, in some situations, forcing fairness increases the throughput, while in other cases using a fairness ($\mathbb{F}$) of less than 1 allows balancing the fairness requirement with the throughput degradation.

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 $IPSw_j$.

[4]It will be shown later that additional forced switches can actually increase the throughput.

3.1. Runtime Calculation of $IPSw_j$

In order to calculate $IPSw_j$ (Equation 9), two thread characteristics must be obtained for each thread: IPM_j and CPM_j. In addition $Miss_{lat}$ 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, 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 (non-resolved misses that are encountered on retirement of the next-to-retire instruction).

The characteristics of the threads can be computed from these counters, as shown in Equation 11, Equation 12 and Equation 13. Average $Miss_{lat}$ can be either a predefined parameter or a measured statistic.

$$IPM_j = \frac{Instrs_j}{\max(Misses_j, 1)} \quad (11)$$

$$CPM_j = \frac{Cycles_j}{\max(Misses_j, 1)} \quad (12)$$

$$IPC_j^{ST} = \frac{IPM_j}{CPM_j + Miss_{lat}} \quad (13)$$

$IPSw_j$ 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

0-7695-2732-9/06 $20.00

an event) or using interrupts. We found $\lambda = 250,000$ cycles to be large enough for our statistical purposes.

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 [37].

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 micro-architecture [19]. The simulator uses Long Instruction Traces (LITs) [38]. 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 [3, 9, 14, 33, 38].

4.1. Machine Configuration

We simulated an out-of-order core, with first level instruction and data caches, a unified 2nd level cache (L2), a pipelined bus and a constant latency memory (as shown in Figure 4). Table 3 summarizes machine configuration parameters. Structure sizes were based on Intel's disclosure of their latest processor core [27], and were slightly increased to reflect our view on a future version of that processor. Memory latency was set to 300 cycles, which is 75ns at 4GHz processor frequency. High level views of the simulated processor pipeline as well as the memory hierarchy are shown in Figure 4.

Fetch/Retire width	4 / 4
RS / ROB size	36 / 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
λ	250,000 cycles

Table 3. Simulated machine parameters

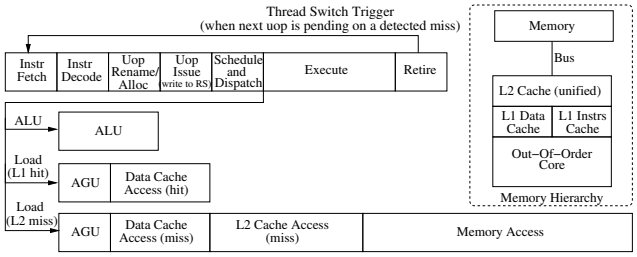

Figure 4. Processor pipeline and memory hierarchy.

We have modified the simulator to support SOE multi-threading, 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 micro-operation to be retired) is flagged as handling a miss which has not been resolved[5]. A thread switch causes draining of instructions from the RS (Reservation Station), ROB and LB (load buffers) and is simulated as a 6 cycles long draining. 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 [31]. 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.

Switch latency is defined as the number of cycles from

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

the start of the switch until the first instruction of the next thread reaches the pipe stage in which the switch was triggered (retirement). The switch latency is not constant (depends for example on the instruction that was switched in), and usually accumulates to around 25 cycles.

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 then 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 [10] were simulated on a two threads 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 micro-architecture 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 16 combinations of benchmarks, out of which 8 combinations were of the same benchmark executed on both threads. Each combination was simulated using SOE without fairness ($\mathbb{F} = 0$), and with $\mathbb{F} = 1, 1/2$ and $1/4$. In addition, we simulated each benchmark alone on the processor, in order to obtain its real IPC_j^{ST}.

5. Simulation Results

5.1. Detailed Examination

We use a representative two threads combination in order to gain insight into the fairness enforcement mechanism. We use a combination of gcc and eon applications. This combination requires active fairness enforcement without which the gcc thread almost starves, while eon runs on the processor most of the time.

5.1.1. Estimating IPC_j^{ST} while Running in SOE. 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 5 (top) shows single thread performance of the two threads when each of them is executed alone on the processor, compared to the

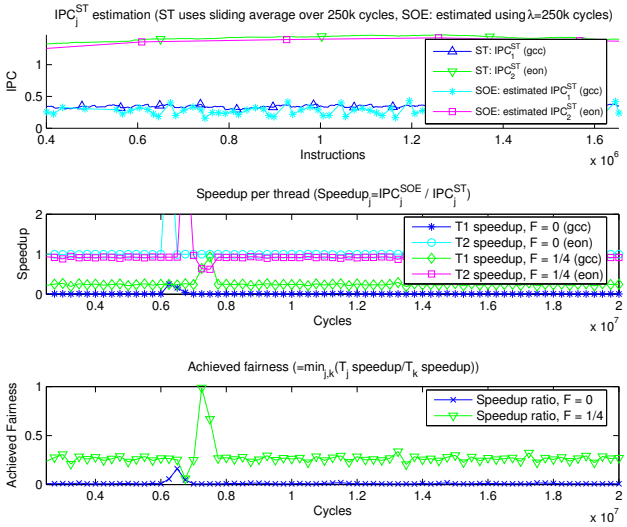

Figure 5. IPC_j^{ST} estimation using hardware counters while running in SOE multithreading with fairness enforcement (top). The estimated speedup of the individual threads when run with and without fairness enforcement (middle). The achieved ratio between thread speedups - achieved fairness (bottom). Fairness is enforced to $1/4$.

estimated performance using hardware counters. Equation 13 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 5 (top), the estimated IPC_j^{ST} closely tracks the real IPC_j^{ST}. It is usually slightly lower than the real IPC_j^{ST}, due to 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.

5.1.2 Fairness Enforcement. Figure 5 (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 2nd thread (eon) executes most of the time, which causes the 1st thread (gcc) to almost starve. When fairness is enforced to $1/4$, the 2nd thread's (eon) speedup is not allowed to exceed the 1st thread's (gcc) speedup by a factor of more than four. Now

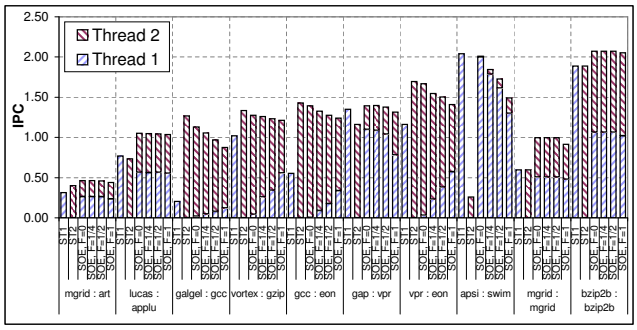

Figure 6. SOE throughput ($IPC^{SOE} = IPC_1^{SOE} + IPC_2^{SOE}$) with and without fairness enforcement, as well as IPC_j^{ST} of each of the two threads when executed alone.

the speed of gcc is 20 times faster than its speed without fairness enforcement.

The speedup plot shows one occurrence in which the 2nd thread gets higher speedup, followed by a slightly higher speedup of the 1st 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 $1/4$. This is most likely due to a short performance phase change, in which the estimation based on the previous λ (250,000) cycles isn't accurate. This causes a short unfair execution. In long runs, however, the effects of these short unfair periods average towards an average of good fairness. The time difference of 1,000,000 cycles indicates that phase change occurred in the 2nd 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 throughput of different thread combinations is shown in Figure 6. IPC^{SOE} with and without fairness enforcement, as well as IPC_j^{ST} for both threads, are shown. IPC^{SOE} is shown as stacked performance of the two threads (Equation 10). The average speedup of SOE over single thread[6] is 24%, 21%, 19% and 15%, for no fairness enforcement ($\mathbb{F} = 0$), $\mathbb{F} = 1/4$, $1/2$ and 1.

Usually, as shown in Figure 6, fairness enforcement has only negligible effect on the throughput when IPC_j^{ST} of the two threads is roughly the same (e.g. *lucas:applu*, *bzip2b:bzip2b*). The greater the difference between performance of the individual threads, the more thread switches will be required in order to enforce fairness. Induced switches cause throughput degradation as enforced fairness $\mathbb{F}$ increases (e.g. *galgel:gcc*, *apsi:swim*).

[6]Speedup of SOE over single thread $= \frac{IPC^{SOE}}{\frac{1}{N}\sum_{j=1}^{N} IPC_j^{ST}} - 1$

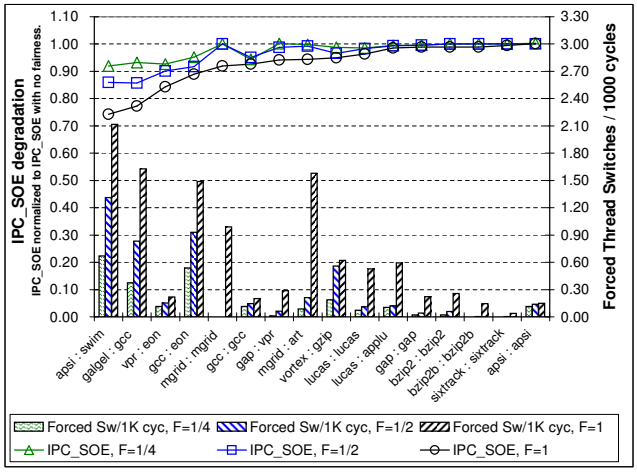

Figure 7. SOE throughput degradation due to fairness enforcement.

Figure 7 shows throughput degradation caused by fairness enforcement, as well as number of forced thread switches in 1000 cycles. It shows throughput normalized to the throughput without fairness enforcement. Forced switches are caused by fairness enforcement (they do not hide any memory access). Enforcing fairness of 1, $1/2$ or $1/4$ causes an average throughput degradation of 7.2%, 3.7% and 2.2% respectively. Enforcing perfect fairness ($\mathbb{F} = 1$) can cause a large performance degradation, even in cases where the same application is executed on both threads. Using fairness of $1/2$ or $1/4$, allows the thread speedups to have some difference (slack), which reduces performance degradation. Throughput degradation is the combined effect of the overhead caused by induced thread switches and the fact that usually slower threads (threads with lower IPC_{no_miss}) are allowed to execute more instructions (at the expense of the faster thread). As shown, there is a high correlation between the number of forced thread switches and the effect on the throughput (performance).

Figure 8 (left) 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 $\mathbb{F} = 1/4$, $1/2$ and 1 show the fairness when it is enforced to be $1/4$, $1/2$ and 1 respectively. Simulation runs are ordered by their achieved fairness when no fairness is enforced. As shown in the figure, even in the most unfair cases fairness enforcement achieves a fairness close to $\mathbb{F}$ (the target fairness being enforced). On runs which are also fair without fairness enforcement, the mechanism has small effect on the achieved fairness.

Figure 8 (right) show the average achieved fairness. It shows the average and standard deviation of $\min(\mathbb{F}, Achieved_{fairness})$. Using $\min(\mathbb{F}, Achieved_{fairness})$ in our calculation ensures

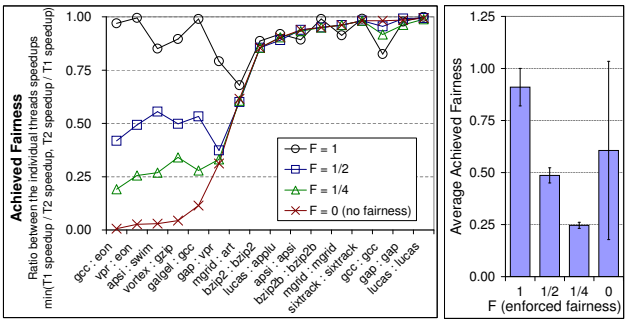

Figure 8. Achieved fairness (left) and its average and standard deviation(right).

that when achieved fairness is greater than $\mathbb{F}$ (the target 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 towards 1). There is no truncation for the case of $\mathbb{F} = 0$. The results show that achieved fairness is highly accurate, although its accuracy degrades as stricter fairness is being enforced.

6. Discussion

Simple heuristics for fairness improvements, such as fetch prioritization or simple time sharing, are ineffective for SOE. Fetch prioritization [29, 41] 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 ineffective for producing high fairness with small performance degradation. When the time sharing quota is small, high fairness can be achieved at the cost of high performance degradation (due to frequent pipeline flushes). When the quota is large, performance is maintained but fair execution is only rarely achieved. Consider the scenario in Example 2. Forcing an average of 400 cycles between thread switches will produce poor fairness. Although time will be equally divided between the two threads, their individual speedups ($\frac{IPC_j^{SOE}}{IPC_j^{ST}}$) will be 0.5 and 0.8, and the achieved fairness will be $\frac{0.5}{0.8} = 0.6$. On the other hand, as shown in Table 2, by using the proposed algorithm the speedup of both threads can be adjusted to 0.63 ($\frac{1}{1.59}$) and the achieved fairness in that simple example will be $\frac{0.63}{0.63} = 1.0$.

Several attempts have been made to suggest a single metric that combines both fairness and throughput. Luo *et al.*[29] used the harmonic mean of the individual threads' speedups. Snavely *et al.*[39] suggested using weighted speedup which is the sum of the individual threads' speedups ($WS = \sum_{i=1}^{N}(IPC_j^{SOE}/IPC_j^{ST})$). We found both metrics to give insufficient insight into either throughput or fairness, and the harmonic mean to be biased toward fairness. We prefer the use of two metrics, one for fairness and one for throughput (IPC). Each of them clearly captures its purpose. Using two metrics allows the user to subjectively balance between fairness and throughput. We believe that fairness should be controlled in such a way that maximum performance is maintained while guaranteeing that no thread starves. This is not possible with either the weighted speedup or the harmonic mean metric. Our fairness metric is based on speedups of the individual threads (similar to the approach used by Luo *et al.*) and clearly encapsulates fairness (but not throughput).

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 negative effects on performance and in many cases on the achieved fairness as well. Both performance 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:gcc or mgrid:mgrid in Figure 8).

Increasing the enforced fairness ($\mathbb{F}$) increases the number of induced thread switches. This increases the total thread switch overhead as well as usually allowing the slower thread to execute more instructions. This usually decreases throughput but increases the achieved fairness. Hence there is a tradeoff between fairness and throughput. The balance point between fairness and throughput depends on the user's preferences. There is no objective metric that combines both throughput and fairness to allow such balance point analysis. According to our simulation results, using $\mathbb{F} \leq 0.5$ is a reasonable compromise. This maintains high throughput and guarantees that no thread starves.

The analytical model shows that enforcing fairness can actually improve throughput. We did not encounter such a case in our simulation runs. Fairness will increase throughput when its enforcement biases execution towards threads that have a higher IPC_{no_miss}. This means that lower CPM threads also have higher IPC_{no_miss}. This occurs for short periods of time, but not enough to increase the average throughput in any of our runs.

The SOE mechsnism 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.

Another example are explicit instructions that can trigger thread switches[7].

We used a constant predefined value for miss latency (300 cycles) to represent the average memory access latency. Some switch events may have variable latency, whose average is hard to predict (e.g. L1 miss). In these cases, 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, $Miss_{lat}$ 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.

7. Conclusions

SOE is a low power and 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 $IPSw_j$ calculation.

Simulation results show that the suggested fairness enforcement mechanism works well for a variety of applications. SOE achieves an average speedup over single thread of 24% when no fairness is enforced. In this case, over a third 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 $\mathbb{F} = 1/4, 1/2$ and 1 for a performance loss of 2.2%, 3.7% and 7.2% respectively. It should be noted that by loosing a very small amount of performance, we *guarantee* thread execution fairness.

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 negligible effect on the execution. Thread switches caused by fairness enforcement usually cause a performance loss, due to the switch latency they incur. Enforcing strict fairness is not necessary, a reasonable compromise is to enforce a fairness of 0.5 or less in order to reduce the impact of thread switches and maintain a high throughput.

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 trivia task due to 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.

[7] An 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.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *Proc. of ISCA-22*, pages 2–13, 1995.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: a processor architecture for multiprocessing. *SIGARCH Comput. Archit. News*, 18(3a):104–114, 1990.

[3] H. Akkary, S. T. Srinivasan, R. Koltur, Y. Patil, and W. Refaai. Perceptron-based branch confidence estimation. In *Proc. of HPCA-10*, page 265, 2004.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proc. of ISCA-32*, pages 506–517, 2005.

[5] C. Bazeghi, F. J. Mesa-Martinez, and J. Renau. uComplexity: Estimating processor design effort. In *Proc. of MICRO-38*, pages 209–218, 2005.

[6] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, 2000.

[7] G. Cai. Power-sensitive multithreaded architecture. In *Proc. of the 2000 IEEE International Conference on Computer Design (ICCD '00)*, page 199, 2000.

[8] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *Proc. of MICRO-37*, pages 171–182, 2004.

[9] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. *SIGOPS Oper. Syst. Rev.*, 36(5):279–290, 2002.

[10] CPU2000. Standard performance evaluation corporation, Spec CPU2000. http://www.spec.org/cpu2000/.

[11] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP throughput with mediocre cores. In *Proc. of PACT-14*, pages 51–62, 2005.

[12] K. J. Duda and D. R. Cheriton. 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*, pages 261–276, 1999.

[13] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, B.-H. Lim, M. S. Squillante, and C.-F. E. Wu. Evaluation of multithreaded processors and thread-switch policies. In *Proc. of the International Symposium on High Performance Computing (ISHPC '97)*, pages 75–90, 1997.

[14] A. Falcon, J. Stark, A. Ramirez, K. Lai, and M. Valero. Prophet/critic hybrid branch prediction. *SIGARCH Comput. Archit. News*, 32(2):250, 2004.

[15] M. K. Farrens and A. R. Pleszkun. Strategies for achieving improved processor throughput. In *Proc. of ISCA-18*, pages 362–369, 1991.

[16] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal*, May 2006.

[17] W. Gruenewald and T. Ungerer. A multithreaded processor designed for distributed shared memory systems. In *Proc. of the 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*, page 206, 1997.

[18] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. *SIGARCH Comput. Archit. News*, 19(3):254–263, 1991.

[19] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2), Febuary 1995.

[20] J. W. Haskins and J. Skadron. 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*, pages 319–328, 2001.

[21] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. *ACM SIGOPS Operating Systems Review*, 31(5):198–211, 1997.

[22] J. Kahle. The Cell processor architecture. In *Proc. of MICRO-38*, page 3, 2005.

[23] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: A dual-core multithreaded processor. In *Proc. of MICRO-37*, pages 40–47, 2004.

[24] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. of PACT-13*, pages 111–122, 2004.

[25] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[26] K. Krewell. AMD vs. Intel in dual-core duel. *Microprocessor Report*, July 2004.

[27] K. Krewell. Intel looks to core for success. *Microprocessor Report*, March 2006.

[28] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[29] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proc. of the International Symposium on Performance Analysis of Systems and Software*, pages 164–171, 2001.

[30] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6, Febuary 2002.

[31] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[32] T. Mowry and S. Ramkissoon. Software-controlled multithreading using informing memory operations. In *Proc. of the 6th International Symposium on High-Performance Computer Architecture (HPCA '00)*, pages 121–132, 2000.

[33] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proc. of HPCA-9*, page 129, 2003.

[34] S. E. Raasch and S. K. Reinhardt. Applications of thread prioritization in SMT processors. In *In Proc. of the Workshop on Multithreaded Execution And Compilation.*, 1999.

[35] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *Proc. of PACT-12*, page 15, 2003.

[36] J. Rattner. Multi-core to the masses. In *Proc. of PACT-14*, page 3, 2005.

[37] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996.

[38] R. Singhal, K. Venkatraman, E. Cohn, J. Holm, D. Koufaty, M.-J. Lin, M. Madhav, M. Mattwandel, N. Nidhi, J. Pearce, and M. Seshadri. Performance analysis and validation of the Intel Pentium4 processor on 90nm technology. *Intel Technology Journal*, 8, February 2004.

[39] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. *Proc. of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 66–76, 2002.

[40] L. Spracklen and S. G. Abraham. Chip multithreading: opportunities and challenges. In *Proc. of HPCA-11*, pages 248–252, 2005.

[41] C. Sun, H. Tang, and M. Zhang. 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*, pages 519– 525, 2005.

[42] J. M. Tendler, J. S. Dodson, J. S. Fields, H. L. Jr., and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[43] R. Thekkath and S. J. Eggers. The effectiveness of multiple hardware contexts. *SIGPLAN Not.*, 29(11):328–337, 1994.

[44] D. M. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proc. of MICRO-34*, pages 318–327, 2001.

[45] D. M. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of ISCA-23*, pages 191–202, 1996.

[46] D. M. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, 1998.

[47] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proc. of MICRO-37*, pages 183–194, 2004.

[48] T. Ungerer, B. Robic, and J. Silc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.