Service Level Agreement for Multithreaded Processors

RON GABOR

Tel Aviv University and Intel Corporation

AVI MENDELSON

Microsoft Corporation

and

SHLOMO WEISS

Tel Aviv University

Multithreading is widely used to increase processor throughput. As the number of shared resources increase, managing them while guaranteeing predicted performance becomes a major problem. Attempts have been made in previous work to ease this via different fairness mechanisms. In this article, we present a new approach to control the resource allocation and sharing via a service level agreement (SLA)-based mechanism; that is, via an agreement in which multithreaded processors guarantee a minimal level of service to the running threads. We introduce a new metric, C_{SLA} , for conformance to SLA in multithreaded processors and show that controlling resources using with SLA allows for higher gains than are achievable by previously suggested fairness techniques. It also permits improving one metric (e.g., power) while maintaining SLA in another (e.g., performance). We compare SLA enforcement to schemes based on other fairness metrics, which are mostly targeted at equalizing execution parameters. We show that using SLA rather than fairness based algorithms provides a range of acceptable execution points from which we can select the point that best fits our optimization target, such as maximizing the weighted speedup (sum of the speedups of the individual threads) or reducing power. We demonstrate the effectiveness of the new SLA approach using switch-on-event (coarse-grained) multithreading. Our weighted speedup improvement scheme successfully enforces SLA while improving the weighted speedup by an average of 10% for unbalanced threads. This result is significant when compared with performance losses that may be incurred by fairness enforcement methods. When optimizing for power reduction in unbalanced threads SLA enforcement reduces the power by an average of 15%. SLA may be complemented by other power reduction methods to achieve further power savings *and* maintain the same service level for the threads. We also demonstrate differentiated SLA, where weighted speedup is maximized while each thread may have a different throughput constraint.

Authors' address: R. Gabor, School of Electrical Engineering, Tel Aviv University; Intel Corporation. Email: rongabor@post.tau.ac.il or ron.gabor@intel.com; A. Mendelson: Microsoft Corporation. Email: avim@microsoft.com; S. Weiss: School of Electrical Engineering, Tel Aviv University. Email: weiss@eng.tau.ac.il.

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 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.
© 2009 ACM 1544-3566/2009/06-ART6 \$10.00

DOI 10.1145/1543753.1543755 <http://doi.acm.org/10.1145/1543753.1543755>

ACM Transactions on Architecture and Code Optimization, Vol. 6, No. 2, Article 6, Publication date: June 2009.

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

General Terms: Design, Performance

Additional Key Words and Phrases: Service level agreement, fairness, performance, throughput, power

ACM Reference Format:

Gabor, R., Mendelson, A., and Weiss, S. 2009. Service level agreement for multithreaded processors. *ACM Trans. Architect. Code Optim.* 6, 2, Article 6 (June 2009), 33 pages.
DOI = 10.1145/1543753.1543755 <http://doi.acm.org/10.1145/1543753.1543755>.

1. INTRODUCTION

Power and manufacturing reasons push the computer industry toward multicore and multithreaded systems. There is an ongoing trend for increasing the number of cores on a chip and the number of threads of a single processor. Processor vendors already supply quad-core processors as mainstream general-purpose processors [Halfhill 2006; McGregor 2007], as well as multithreaded eight-core processors for servers and high performance computing [Kongetira et al. 2005]. Some vendors have also announced plans for “many-core” products with a large number of cores integrated on a single chip [Pawlowski 2007]. Highly threaded processors and virtual machines are used for server consolidation [Rosenblum and Garfinkel 2005]. These servers can run a number of independent threads from different users. The higher the number of threads sharing a resource, the higher the probability of imbalance in the resource usage [Cazorla et al. 2004; Gabor et al. 2007; Raasch and Reinhardt 1999]. For example, in a multicore processor, one thread may hog the memory bandwidth, causing other threads to run very slowly. In multithreaded processors, where the pipelines and other resources are shared, one thread may use most of a shared resource, such as the data cache, evicting the data of the other threads and causing them to severely slow down. Fairness enforcement [Cazorla et al. 2004; Gabor et al. 2006; Raasch and Reinhardt 2003] was proposed in order to ease the effect of such imbalance resource allocation, but fairness does not always work as well as expected, as we will show later in this article.

Resources are shared between threads in order to achieve high power and area utilization. As the number of shared resources increases, the resource allocation problem becomes more severe. For example, in SMT (Simultaneous Multi-Threading), where the pipelines and other structures are shared, special fairness mechanisms need to be implemented in order to achieve predictable performance. In CMP (Chip Multi Processing), last level caches, buses and power consumption are shared, and usually require fairness mechanisms [Chang and Sohi 2007; Hsu et al. 2006; Iyer 2004; Iyer et al. 2007; Kim et al. 2004; Rafique et al. 2006; Yeh and Reinman 2005]. SOE (Switch-on-Event or coarse-grained) multithreading is an in-between multithreading solution, it allows sharing all the resources of the processor in a timeshared mode. SOE is an attractive multithreading approach that allows us to use the analytical framework developed by Gabor et al. [2006, 2007] for this work.

Admission control [Guo et al. 2007] can be used to reduce fairness issues by allowing or forbidding jobs from running together. A different approach to fairness issues is to implement fairness mechanisms into the processor to enforce fairness among the running threads. These mechanisms include, for example, resource allocation and partitioning, and complement admission control.

Previous work on chip resource allocation usually focused on achieving fairness of allocation between the different threads. In this article, we propose a new approach, based on Service Level Agreement (SLA); that is, based on an agreement in which multithreaded processors guarantee a minimal level of service to the running threads. The use of SLA allows us to shift the notion of “fair systems” to “balanced systems.” The balance between the resources is based on the demand, or requirement of the resources, as viewed by the system at any given time. The system maintains a minimum level of service for these resources for each thread and divides the leftover between the threads so that optimal execution is achieved.

SLA acts as a contract specifying the minimal service level that is acceptable by each thread. It can relate to different parameters, such as performance, resource allocation and even to power and thermal budgets. Service level can be quantified using the ratio of the measured parameter value when threads are executing together over their single thread execution value. For example, performance service level can be quantified by using the speedups of the individual threads and service level in shared caches can be quantified by using the decrease in hit rate or decrease in cache share caused by the sharing of the cache. The case when all of the threads experience similar degradation in the measured parameter (due to the sharing) is usually considered fair. For example, fairness in multithreaded processors can be considered as equal speedups to all threads [Gabor et al. 2006; Kondo et al. 2007; Luo et al. 2001]. This case shows that fairness does not necessarily provide conformance to SLA (all threads getting an equal nonacceptable service level is considered fair). Conforming to SLA, however, provides each thread with a minimal service level, which ensures that no thread is hogging any resource or is causing a starvation of any other thread.

This article presents a new metric, C_{SLA} , that measures conformance to SLA. The C_{SLA} metric is different from fairness metrics, which attempt to equalize parameters of different threads. The new C_{SLA} metric is suitable for use as a constraint on the execution while other metrics are being maximized or minimized. For example, we demonstrate in this article that our C_{SLA} metric is suitable for being used as a constraint while maximizing the weighted speedup¹ (*WeightedSpeedup*), achieving higher *WeightedSpeedup* than the *WeightedSpeedup* achieved using fairness metrics. We also demonstrate that C_{SLA} may be used as a constraint on the speedups of the individual threads while reducing the overall average power.

¹Weighted speedup (*WeightedSpeedup*) is the sum of the speedups of the individual threads [Snively and Tullsen 2000; Tullsen and Brown 2001].

Table I. Abbreviations and Symbols

(a) General ($p_j = \text{Speedup}_j$)

Symbol	Description
IPC_j^{ST}	IPC of thread j when executed alone.
IPC_j^{MT}	IPC of thread j when executed together with other threads.
$Speedup_j$	speedup of thread j : $Speedup_j = (IPC_j^{MT} / IPC_j^{ST})$
$WeightedSpeedup$	Weighted speedup: $WeightedSpeedup = \sum_{j=1}^N Speedup_j$
$HarmonicMean$	Harmonic mean of speedups: $HarmonicMean = \frac{N}{\sum_{j=1}^N \frac{1}{Speedup_j}}$
F_{stddev}	Fairness metric based on standard deviation of a parameter
F_{var}	Fairness metric based on the Variance of a parameter
$F_{max-min}$	Max-Min difference fairness metric: $F_{max-min} = \max_j(Speedup_j) - \min_j(Speedup_j)$
$F_{\frac{min}{max}}$	Fairness metric based on Max-Min ratio: $F_{\frac{min}{max}} = \frac{\min_j(Speedup_j)}{\max_j(Speedup_j)}$
C_{SLA}	Conformance to service level agreement (1)

SOE

Symbol	Description
IPC_j^{SOE}	IPC of thread j when executed using SOE with other threads.
IPC_{no_miss}	IPC of a thread excluding last-level misses (as if the last level cache always hits).
$InstPerMiss_j$	Instructions per miss in thread j . The average number of instructions between two consecutive misses in thread j .
$CycPerMiss_j$	Cycles per miss in thread j . The average number of cycles between two consecutive misses in a thread j .
$InstPerSwitch_j$	Instructions per switch in thread j . The average number of instructions thread j executes before it is switched out (in SOE).
$CycPerSwitch_j$	Cycles per switch in thread j . The average number of cycles thread j executes before it is switched out (in SOE).

The contributions of this article are as follows:

- We introduce a novel metric for measuring the Conformance to Service Level Agreement, C_{SLA} , for multithreaded processors. This metric is applicable to any multithreaded or multicore system and for measuring conformance to SLA in any measured parameter (performance, resource sharing, etc.).
- We compare the new C_{SLA} metric to previously suggested fairness metrics, such as F_{stddev} , $F_{max-min}$, $F_{\frac{min}{max}}$ and $HarmonicMean$ (see Table I). This comparison gives insight on the usefulness of each metric.
- We show examples of using C_{SLA} for setting a constraint on the execution, which provides a range of acceptable execution points allowing for the optimization of the execution. We use C_{SLA} based on the speedups of the individual threads to maximize the weighted speedup or reduce the average power, while satisfying the SLA requirements.
- We demonstrate the effectiveness of the SLA approach in a differentiated service case, in which the weighted speedup is maximized while each thread may have a different throughput constraint.
- We study the new C_{SLA} metric, in the context of switch-on-event (SOE) multithreading and show how the weighted speedup can be increased beyond that

which is achieved while using traditional fairness approaches, and how to apply the C_{SLA} metric to reduce the power.

The rest of this article is organized as follows. Section 2 describes previously suggested fairness metrics. Section 3 presents the new conformance to service level agreement metric (C_{SLA}). Section 4 describes the inherent SLA issues in switch-on-event (SOE) multithreading and presents the SOE analytical model. Section 5 is an extended example. Sections 6 and 7 show how to maximize the weighted speedup and how to reduce the power while satisfying SLA requirements in a two-thread SOE model, as well as empirical results. Section 8 demonstrates the effectiveness of SLA approach in differentiated service case, where each thread may have a different throughput constraint. Section 9 describes simulation tools and methodology including machine configuration and SOE implementation details. Section 10 describes related work. We conclude with the summary in Section 11.

2. FAIRNESS METRICS

SLA guarantees that each thread receives the required level of service. We will define a metric for measuring the conformance to SLA in Section 3. Several metrics have been suggested for the quantification of fairness in multithreaded processors. This section describes these metrics. In the following sections, we will compare SLA enforcement to fairness enforcement.

Admission control [Guo et al. 2007] attempts to decide which threads should be allowed to run together, before they are executed, in order to reduce mutual interference effects. For the purpose of admission control, absolute metrics (e.g., cache or bandwidth absolute stated requirement) are intuitive and useful. Absolute metrics can be easily summed up and used for admission decisions. On the other hand, SLA and fairness measure the threads mutual effects while they are running, and are usually measured as relative numbers compared to a baseline (e.g., single-threaded execution parameters or even relative to the stated requirement). Fairness and SLA can complement admission control, by using the relative metrics to determine if all threads are treated fairly. This is important when resources are insufficient (e.g., resource oversubscription), which can still happen even when using admission control. SLA can also be used to decide how to divide the excess, in order to maximize or minimize some target (e.g., maximize performance or reduce power).

SLA and fairness can relate to different parameters such as performance, resource allocation, and even to power and thermal budgets. Fairness is usually quantified using the ratio of the measured parameter value when threads are executing together over their single thread execution value. Let p_j^{MT} and p_j^{ST} be the measured parameter value of thread j when executed together with other threads (MT) and alone (ST). We define p_j as $\frac{p_j^{MT}}{p_j^{ST}}$. Using this notation we can define, for example, fairness or service level in throughput using the IPC of the different threads. In this case, the fairness or service level will be defined using the speedups of the individual threads: $p_j = \text{Speedup}_j = \frac{IPC_j^{MT}}{IPC_j^{ST}}$ (see Table I for abbreviations and symbols used in this article).

Fig. 1. Different metrics behavior in 2D space. Brighter color denotes higher value (black is 0; white is 1).

Several metrics have been proposed for fairness quantification in microprocessors.

Variance (F_{var}) and Stdev (F_{stdev}) Fairness can be quantified using the variance or the standard deviation of p_j [Hsu et al. 2006; Luo et al. 2001]. A fair situation is considered when the variance or the standard deviation is minimized, and absolute (perfect) fairness is when $p_1=p_2=\dots=p_n$, which results in $F_{var}=F_{stdev}=0$. For example, when performance is considered, fairness can be quantified using the standard deviation of the speedups of the individual threads, and absolute fairness is when all threads experience the same speedup. Since variance and standard deviation are conceptually similar, we will use only the standard deviation in the rest of the article.

Max-Min Difference ($F_{max-min}$) Fairness can be quantified using the difference between the maximum and minimum of p_j : $F_{max-min} = \max_j(p_j) - \min_j(p_j)$. The lower the value, the higher the fairness, and a value of 0 is absolute fairness ($p_1=p_2=\dots=p_n$) [Kim et al. 2004; Kondo et al. 2007].

Max-Min Ratio ($F_{min_{max}}$) Fairness can be quantified using the ratio between the minimum and the maximum of p_j : $F_{min_{max}} = \frac{\min_j(p_j)}{\max_j(p_j)}$. A value of 1 is absolute fairness ($p_1=p_2=\dots=p_n$), and 0 means that at least one of the threads is starved [Gabor et al. 2006; Chen and Ma 2007].

Harmonic Mean (HarmonicMean) The Harmonic Mean of p_j can be used to encapsulate fairness as well as the goodness of p_j into a single metric: $HarmonicMean(p_j) = \frac{N}{\sum_{j=1}^N (\frac{1}{p_j})}$ [Cazorla et al. 2003; Luo et al. 2001]. For example, it can be used to encapsulate both the fairness in throughput (IPCs of the individual threads) and the achieved speedups (throughput improvement) into a single metric.

Figure 1 shows how the different metrics behave in the case of two threads, for the case of $0 \leq p_j \leq 1$. F_{stdev} (Figure 1(a)) and $F_{max-min}$ (Figure 1(b)) behave in a similar manner, except that standard deviation is always higher than the max-min difference. Both F_{stdev} and $F_{max-min}$ achieve highest fairness when $p_1=p_2=\dots=p_n$, in which case $F_{stdev}=F_{max-min}=0$. The difference between these two metrics is when more than two threads are used, in which case the standard deviation averages the fairness over all threads while the max-min difference ($F_{max-min}$) uses the worst-case difference. Figure 1(c) shows $F_{min_{max}}$, which also achieves its highest fairness when $p_1=p_2=\dots=p_n$ (in which case $F_{min_{max}}=1$). $F_{min_{max}}$ considers fairness from a worst-case scenario, using the two threads that achieve the max and min values of p_j . The *HarmonicMean* metric (Figure 1(d))

achieves its highest value when $p_1=p_2=\dots=p_n=\max_j(p_j)$, and its value decreases as we get further away from that point. The *HarmonicMean* metric attempts to encapsulate both the fairness of p_j and its value into a single metric.

Jain et al. [1998] defined desired properties of fairness metrics: (i) Population size independence: fairness metrics should not depend on the number of threads in the processor. This requirement is satisfied by all of the previously described metrics. (ii) Scale and metric independence: fairness metric should not depend on the measured parameter or on its scale. This requirement is satisfied only by $F_{\frac{\min}{\max}}$.² (iii) Boundedness: fairness metric should be bounded between 0 and 1. This is useful for the definition of discrimination metrics and enables intuitive use of the metric value. F_{stddev} and $F_{\text{max-min}}$ have a minimum value of 0 (best fairness) but do not have an upper bound.³ $F_{\frac{\min}{\max}}$ is always bound between 0 and 1 (as required) and *HarmonicMean* has no upper bound (but *HarmonicMean* is a combined measure of performance and fairness, so it can deviate from this requirement). (iv) Continuity: a change in system fairness should be reflected in a continuous manner in the metric. All of the metrics satisfy this requirement. $F_{\frac{\min}{\max}}$ is the only metric that satisfies all of et al. [Jain et al. 1998] Jain et al. [1998] requirements for fairness metrics.

Jain et al. also defined another, somewhat subjective, requirement—intuitive for use. The metrics’ value should have a clear, intuitive and easy to use meaning. From our experience, F_{stddev} and $F_{\text{max-min}}$ are not intuitive for use. For example, $F_{\text{max-min}}=0.2$ when measuring fairness of IPC, may mean that one thread has starved, while the other achieved 0.2 of its IPC^{ST} (the IPC of when it was executing alone). $F_{\text{max-min}}=0.2$ may, on the other hand, mean that one thread achieved its full IPC^{ST} , while the other thread achieved 0.8 of its IPC^{ST} . From our perspective of fairness, the former scenario is by far worse than the latter. $F_{\frac{\min}{\max}}$ is more intuitive than $F_{\text{max-min}}$, since it is bound between 0 and 1. $F_{\frac{\min}{\max}}$, however, may also present similar dilemmas as $F_{\text{max-min}}$. For example $F_{\frac{\min}{\max}}=0.5$, when measuring fairness of IPC , may mean that two threads achieved 0.01 and 0.02 of their IPC^{ST} (lost 99% and 98% of their performance), but it can also mean that one thread maintained its full performance while the other lost 50%.

All of the previously described fairness metrics, with the exception of *HarmonicMean*, consider a system as fair whenever all of the threads have the same value of $p_j = \frac{p_j^{MT}}{p_j^{ST}}$. This sounds reasonable, but it may lead to strange effects when it is being enforced. Let us apply the fairness metrics to cache usage in a dual-core processor with a shared cache. Let us assume we run two independent threads. The first thread is a large matrix calculation, whose dataset fits exactly in the cache. The second thread is a small matrix calculation whose

²Let us consider, for example, the case of measuring throughput using IPC (instructions per cycle) or using IPS (instructions per second). IPC and IPS measure the throughput using different scales ($IPS=IPC \cdot Freq$). The value of F_{stddev} , $F_{\text{max-min}}$ and *HarmonicMean* will depend on the scale.

³There is no guarantee that $p_j = \frac{p_j^{MT}}{p_j^{ST}} \leq 1$. Even in the case of IPC , an upper bound is not necessarily 1, since there may be cases where one of the threads will have a speedup >1 when the two threads are running together (e.g., when the other threads acts intentionally or unintentionally as a prefetcher, helping the first thread to achieve higher IPC when the two are executed together).

dataset fits within a small part of the cache. Before we can use any fairness metric, we have to define p_j . In the case of cache usage, p_j can measure, for example, cache share, cache hit rate, or hit count. Assuming we use hit rate as p_j , the first thread, whose dataset size is the same as the cache size, will suffer a large decrease in its hit rate due to the sharing. This happens because its hit rate when it was running alone was 1 (always hit), but when the other thread takes up part of the cache, the matrix no longer fits, and each miss causes some other useful part of the matrix to be evicted. In order to achieve high fairness using F_{stdev} , $F_{\text{max-min}}$ or $F_{\frac{\min}{\max}}$, the degradation in hit rate should be the same for both threads. This means that since the first thread has high degradation in its hit rate, the other thread's hit rate should also severely degrade. Enforcing fairness, in this example, based on equalizing p_j harms one of the threads with no or only very little benefit for the other.

The previously described example illustrates the shortcoming of any fairness metric that attempts to equalize the degradation (or change) caused by the sharing. Furthermore, from a user's perspective, equalizing p_j can lead to strange behavior. For example, consider a dual-threaded processor. An application of a certain user may lose most of its performance because of another user's application. This situation may still be considered fair if the other application also lost the same amount of performance. From the user's perspective, a fair situation is when the sharing of the dual-threaded processor does not degrade the performance by more than 2, otherwise it would be better to simply run the applications one after the other.

It should be noted that fairness is not limited to parameters whose value represent "goodness" (where high value is good), such as IPC or cache hit rate. Fairness can also apply to parameters where a lower value represents goodness, such as CPI or miss rate. In order to simplify the discussion, we assume that in such cases $\frac{1}{p_j}$ is used instead of p_j .

3. CONFORMANCE TO SERVICE LEVEL AGREEMENT METRIC (C_{SLA})

The shortcomings of the various fairness metrics previously described lead us to the conclusion that there is a need for an alternative, a metric that would encapsulate the user's perspective of sharing resources and degradation in measured parameters. Hence we suggest a different approach, service level agreement (SLA). A system satisfies SLA as long as all the threads achieve a value of p_j that is higher or equal to a predefined requirement, R .

The conformance to SLA metric, C_{SLA} , can be formally defined as shown in (1).

$$C_{\text{SLA}} = \begin{cases} \min_j(p_j) \cdot N & \text{if } \min_j(p_j) < 1/N \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

where N is the number of threads and the required value is $R = 1/N$.

The definition of C_{SLA} allows comparing threads based on how close they are to the required level of p_j . The thread that experiences the worst degradation determines the overall metric value. C_{SLA} saturates at 1, which means that the requirement is satisfied by all of the threads. The minimal value of C_{SLA} is 0, which is the lowest conformance value (at least one of the threads has $p_j=0$).

Note that C_{SLA} is the system conformance level; if one of the threads has $p_j=0$, then $C_{SLA}=0$ even if other threads receive service at or above the required level.

In a dual-threaded processor, we can define the performance (IPC based) SLA using a required value of $R=1/2$. In this case, SLA is maintained as long as $Speedup_j = \frac{IPC_j^{MT}}{IPC_j^{ST}} \geq \frac{1}{2}$ for both threads. Figure 1(e) shows how C_{SLA} behaves in a two-thread scenario, with $R=1/2$. As shown, as long as both threads have their $p_j \geq R$, the value of C_{SLA} is 1 (shown as white). The value of the C_{SLA} metric decreases when p_j of any thread goes below $R=1/2$.

Sections 6, 7, and 8 demonstrate the effectiveness of using $C_{SLA}=1$ as a constraint when optimizing execution or reducing the power in SOE. Our approach and the C_{SLA} metric are by no means limited to SOE and are also applicable to CMP and SMT.

In order to use C_{SLA} , a required value (R —the constraint) needs to be specified. This can be done, for example, by the system administrator, who should also specify the target execution optimization when the requirement is met by all threads (e.g., minimizing the power or maximizing the performance). When the requirement can not be met, the system should attempt to maximize the C_{SLA} , which means get the worst-case thread as close as possible to the requirement. This usually means equalizing the measured metric degradation over all threads (fair execution). The use of relative requirement is useful when threads characteristics are unknown or have large variations. It is also useful when determining the relative degradation one is willing to tolerate in order to optimize the execution (e.g., tolerate some performance degradation in order to reduce power).

4. SLA AND FAIRNESS IN SWITCH ON EVENT MULTITHREADING

In this section, we analyze the conformance level and fairness in Switch-On-Event Multithreading (SOE) using an analytical model. In SOE, the core runs a single thread at any given time. When the thread is expected to stall for a long time (e.g., on a last level cache miss) the thread is swapped out and replaced with another. Switching threads is done using a pipeline flush [Eickemeyer et al. 1997; Farrens and Pleszkun 1991; Thekkath and Eggers 1994]. From a software perspective, the core contains multiple threads that run concurrently. The core, however, multiplexes the threads using predefined detectable events, such as long-latency miss events. SOE has been implemented in several multithreaded commercial processors, such as IBM's RS64 IV [Borkenhagen et al. 2000] and Intel's Montecito [McNairy and Bhatia 2005] (both of them implementing two-thread SOE).

Gabor et al. [2006] describe the fairness problem in SOE, when threads with different *CycPerMiss* (average cycles between misses or long stall events) are executed together. Let us consider a two-thread SOE processor switching on last-level cache misses. The fairness problem arises when one of the executed threads has a lower miss frequency than the other. Figure 2 shows this case where $T1$ has a much lower miss frequency (higher average number of cycles between misses) than $T2$. In this case, the thread with the high miss frequency ($T2$) switches out quickly, giving way to the low miss-frequency thread ($T1$).

Fig. 2. Example of unfair execution in SOE. $T1$ has lower miss frequency (higher average number of cycles between misses) than $T2$, which leads to $T1$ getting more execution time than $T2$.

Fig. 3. Representation of single-thread runs (top) and SOE (bottom).

The low miss-frequency thread ($T1$) will then execute for a longer time, until it switches out. This kind of round robin between the two threads produces the situation where the low miss-frequency thread ($T1$) gets more cycles to run than the high miss-frequency thread ($T2$), leading to unfair execution, which may not be of an acceptable SLA.

We use the Gabor et al. [Gabor et al. 2006; Gabor et al. 2007] SOE analytical model, which represents the execution of single-thread applications as sequences of instructions delimited by long latency stalls (last-level cache misses). *InstPerMiss* (instructions per miss) is the average number of useful instructions executed between two consecutive misses, and *CycPerMiss* (cycles per miss) is the average number of cycles between those misses. When a thread executes alone on the processor, each miss causes an execution stall of *MissLatency* cycles (*MissLatency* is the average memory access time in processor cycles). *InstPerMiss_j* and *CycPerMiss_j* (*InstPerMiss* and *CycPerMiss* of thread j) are illustrated in Figure 3 for two threads, both when running together (using SOE), and when executed separately (alone).

We can use *InstPerMiss_j* and *CycPerMiss_j* to analyze the *IPCs*, speedups, weighted speedup, conformance level and the fairness using different approaches. The *IPC* of each thread in single-thread mode and in SOE mode is shown in Eq. (2) and Eq. (3), where *SwitchLatency* is the average number of cycles a thread switch takes in SOE.

$$IPC_j^{ST} = \frac{InstPerMiss_j}{CycPerMiss_j + MissLatency} \quad (2)$$

$$IPC_j^{SOE} = \frac{InstPerMiss_j}{\sum_{k=1}^N (CycPerMiss_k + SwitchLatency)} \quad (3)$$

This representation is based on several assumptions. It has been assumed that the work done speculatively while misses are being resolved is relatively small and that misses do not overlap. It has been also assumed that resource sharing (e.g., branch prediction history tables sharing) has a negligible impact on the model accuracy. These assumptions simplify the model and do not significantly reduce its accuracy [Gabor et al. 2006]. In order to work around the miss overlapping effect, we limit our analysis in the rest of the paper to the case when $CycPerMiss_j \geq MissLatency - 2SwitchLatency$ (in a two-thread SOE model), a reasonable assumption holding in all of the cases we analyzed. The model uses round robin thread switching, which is reasonable for an SOE implementation [Gabor et al. 2007].

To enforce fairness or SLA in SOE, we induce additional thread switches, not necessarily hiding any long stall event. The use of induced switches introduces two variables: $CycPerSwitch_j$ and $InstPerSwitch_j$, which are the average number of cycles and instructions between switches in thread j . In this case, the IPC_j^{SOE} is defined in (4).

$$IPC_j^{SOE} = \frac{InstPerSwitch_j}{\sum_{k=1}^N (CycPerSwitch_k + SwitchLatency)} \quad (4)$$

Using Eq. (2) and Eq. (4) we can express $Speedup_j$, the speedup of thread j , as shown in Eq. (5):

$$Speedup_j = \frac{IPC_j^{SOE}}{IPC_j^{ST}} \quad (5)$$

Eq. (5) and Eq. (4) can be used to calculate different metrics, such as aggregate IPC, weighted speedup (*WeightedSpeedup*), harmonic mean of speedups (*HarmonicMean*), as well as fairness metrics, such as $F_{\min}^{\max}$ [Gabor et al. 2006, 2007]. C_{SLA} can be calculated as defined in Eq. (1).

$$Aggregate\ IPC = \sum_{j=1}^N IPC_j^{SOE} \quad (6)$$

$$WeightedSpeedup = \sum_{j=1}^N Speedup_j \quad (7)$$

$$HarmonicMean = \frac{N}{\sum_{j=1}^N \frac{1}{Speedup_j}} \quad (8)$$

$$F_{\min}^{\max} = \frac{\min_j(Speedup_j)}{\max_j(Speedup_j)} \quad (9)$$

5. EXAMPLE: SOE WITH DIFFERENT OPTIMIZATION TARGETS

In this section, we provide an example that illustrates the concepts presented so far.⁴ In the example, we use the analytical model described in the previous

⁴Skipping this section does not affect the readability of the rest of this article.

Table II. Example of Two-Thread SOE with Different Optimization Targets ($MissLatency=300$, $SwitchLatency=25$)

	Plain SOE		Optimized for Agg IPC		Optimized for Agg IPC & SLA		Optimized for $F_{\min}^{\max}$	
$InstPerMiss_j$	600	3,500	600	3,500	600	3,500	600	3,500
IPC_{no_miss}	1.0	2.5	1.0	2.5	1.0	2.5	1.0	
$InstPerSwitch_j$	600	3,500	250	3,500	600	2,875	600	1,853
$CycPerSwitch_j$	600	1,400	250	1,400	600	1,150	600	741
IPC_j^{ST}	0.667	2.059	0.667	2.059	0.667	2.059	0.667	2.059
IPC_j^{SOE}	0.293	1.707	0.147	2.059	0.333	1.597	0.431	1.332
$\frac{IPC_j^{SOE}}{IPC_j^{ST}}$	0.439	0.829	0.221	1.000	0.500	0.776	0.647	0.647
$F_{\min}^{\max}$	0.529		0.221		0.645		1.000	
$HarmonicMean$	0.574		0.361		0.608		0.647	
C_{SLA}	0.878		0.441		1.000		1.000	
$WeightedSpeedup$	1.268		1.221		1.276		1.294	
Agg IPC	2.000		2.206		1.931		1.763	

	Optimized for $HarmonicMean$		Optimized for $WeightedSpeedup$ & SLA		Optimized for $WeightedSpeedup$	
$InstPerMiss_j$	600	3,500	600	3,500	600	3,500
IPC_{no_miss}	1.0	2.5	1.0	2.5	1.0	2.5
$InstPerSwitch_j$	600	1,735	600	1,138	600	625
$CycPerSwitch_j$	600	694	600	455	600	250
IPC_j^{ST}	0.667	2.059	0.667	2.059	0.667	2.059
IPC_j^{SOE}	0.446	1.291	0.543	1.029	0.667	0.694
$\frac{IPC_j^{SOE}}{IPC_j^{ST}}$	0.670	0.627	0.814	0.500	1.000	0.337
$F_{\min}^{\max}$	0.529		0.614		0.337	
$HarmonicMean$	0.574		0.620		0.504	
C_{SLA}	0.878		1.000		0.675	
$WeightedSpeedup$	1.268		1.314		1.337	
Agg IPC	2.000		1.572		1.361	

section and show that maximizing the weighted speedup or the aggregate IPC does not necessarily achieve an acceptable level of service. Optimizing the execution for SLA, however, increases the weighted speedup, and may increase or decrease the aggregate IPC (depending on the IPCs of the individual threads).

We consider a two-thread case, as shown in Table II. The single-thread behavior of each thread is defined by its $InstPerMiss_j$ and ideal IPC (IPC_{no_miss}) given in the table. The table shows plain SOE in which thread switches are caused only by misses, as well as SOE runs in which $CycPerSwitch_j$ was set to maximize one of the metrics. Optimizing for aggregate IPC means setting $CycPerSwitch_j$ to maximize $\sum_j IPC_j^{SOE}$; optimizing for “aggregate IPC & SLA” maximizes the aggregate IPC under the constraint of $C_{SLA}=1$ (conforming to SLA); optimizing for $F_{\min}^{\max}$ (which is the same as optimizing for $F_{\max-min}$ or F_{stdev}) means equalizing the speedups of both threads; optimizing for $HarmonicMean$

Fig. 4. Comparison of the different metrics in the example.

(harmonic mean of speedups) maximizes the combined throughput and fairness metric of Luo et al. [2001]; optimizing for *WeightedSpeedup* maximizes the weighted speedup (with or without the constraint of $C_{SLA}=1$).

Figure 4 shows the results of the different optimization points. The vertical axis is the achieved value of the five metrics. The values of the five metrics for the different optimization points are groups on the horizontal axis. Plain SOE is the baseline, which achieves a low degree of fairness and C_{SLA} ($F_{\min/\max}=0.5$; $C_{SLA}=0.88$). Optimizing for aggregate IPC yields the lowest fairness, C_{SLA} and *WeightedSpeedup* when compared to any other optimization point, including plain SOE. This is because optimizing for aggregate IPC means giving the thread with higher $IPC_{no_miss}(T_2)$ as much time to run as possible, switching to the other thread for the duration of any miss handling, and switching back as soon as the miss is resolved. Optimizing the execution for *WeightedSpeedup* also yields low fairness and C_{SLA} , since the execution is biased strongly toward the thread with lower $CycPerMiss(T_1)$, allowing the other thread to run only while misses in T_1 are handled. In this example, optimizing for *WeightedSpeedup* reduces the aggregate IPC and vice versa (we will show later that this is not always the case).

Optimizing the execution for $F_{\min/\max}$ results also in achieving high C_{SLA} , as well as high *HarmonicMean* and *WeightedSpeedup* (higher than plain SOE). Optimizing for aggregate IPC or *WeightedSpeedup* with SLA (under the constraint of $C_{SLA}=1$) achieves nearly the same results as when optimizing with no constraints at all. When SLA is enforced the optimization yields acceptable execution (as defined by C_{SLA}) and improves $F_{\min/\max}$ relative to plain SOE.

The example shows that maximizing aggregate IPC or the *WeightedSpeedup* does not necessarily result in fair execution. Enforcing fairness using $F_{\min/\max}$

Fig. 5. Aggregate IPC and *WeightedSpeedup* (WS) space exploration. $F_{\min}^{\max}=1$ shown as a dark stripe and $C_{SLA}=1$ as a wider lighter stripe. $CPSw$ is *CycPerSwitch* and CPM is *CycPerMiss*.

improves the *WeightedSpeedup* but causes a decrease in aggregate IPC (since it biased the execution toward the lower IPC_{no_miss} thread). On the other hand, enforcing SLA using the $C_{SLA}=1$ constraint increases *WeightedSpeedup* and causes only a slight decrease in the aggregate IPC.

We can also examine the various runs of the example as points in the two-dimensional space of switching points $CycPerSwitch_j$. $CycPerSwitch_j$ ranges from $MissLatency - 2SwitchLatency$ to $CycPerMiss$. Figures 5(a) shows the aggregate IPC as a function of $CycPerSwitch_1$ and $CycPerSwitch_2$, with two annotated points, plain SOE and the maximum aggregate IPC point (shown in Table II as “Optimized for Agg IPC”). The thin dark stripe marks $F_{\min}^{\max}=1$ (where the speedups of the individual threads are equalized), the wider stripe shows where $C_{SLA}=1$ (the speedup of the individual threads is $\geq 1/2$), and the rest is where $C_{SLA} < 1$. Figure 5(b) shows the same space for *WeightedSpeedup*.

As shown, the $C_{SLA}=1$ constraint is less “strict” (meaning more points satisfy the constraint) than $F_{\min}^{\max}=1$. The larger area in the graph covered by $C_{SLA}=1$, relative to $F_{\min}^{\max}=1$, allows the selection of points with higher aggregate IPC or *WeightedSpeedup*. Plain SOE as well as the points with maximum aggregate IPC or maximum *WeightedSpeedup* have unacceptable values of C_{SLA} and fairness ($C_{SLA} < 1$; $F_{\min}^{\max}$ too low).

In this example, enforcing fairness or SLA ($F_{\min}^{\max}=1$ or $C_{SLA}=1$) increases the *WeightedSpeedup* but reduces the aggregate IPC. Had the IPC_{no_miss} of the two threads been switched (so that $IPC_{no_miss}=2.5$ and 1.0 for T_1 and T_2 , respectively), then optimizing for aggregate IPC or for *WeightedSpeedup* (with or without the constraint of $C_{SLA}=1$) would have increased the aggregate IPC over that of plain SOE. This is so because, in this case, the low *CycPerMiss* thread (which is favored when optimizing for *WeightedSpeedup*) is the same thread that has the higher IPC_{no_miss} (which is favored when optimizing for aggregate IPC). In this case, optimizing for aggregate IPC or for *WeightedSpeedup* with the constraint of $C_{SLA}=1$ would result in the same execution parameters (improving *WeightedSpeedup* and aggregate IPC over plain SOE). Optimizing for $F_{\min}^{\max}$ would also improve the aggregate IPC because it would also bias the execution toward the high IPC thread. This case shows that enforcing fairness may improve the aggregate IPC.

Fig. 6. Achieved conformance to the SLA as measured by C_{SLA} .

6. MAXIMIZING THE WEIGHTED SPEEDUP UNDER SLA

When enforcing SLA, we can choose to optimize performance by maximizing either the aggregate IPC or the *WeightedSpeedup*. The two optimization targets produce different results in most cases. In this section, we demonstrate the effectiveness of using the C_{SLA} metric for maximizing the weighted speedup. A similar method can be used with aggregate IPC as the target.

We simulated a two-thread SOE processor, as described in Section 9. In each thread, the number of misses causing a switch and the number of cycles up to the switch are counted. These counters are then used to compute the *CycPerMiss* of each thread every 250,000 cycles. The results are used to calculate $CycPerSwitch_j$ that needs to be maintained in order to maximize *WeightedSpeedup* under SLA (see Appendix A for more details). This $CycPerSwitch_j$ is enforced in the following 250,000 cycles using a deficit mechanism. A thread switches out when its cycle quota (based on $CycPerSwitch_j$) elapses without a miss. The deficit mechanism keeps track of the $CycPerSwitch_j$ quota left over when a miss triggers a thread switch. The left over quota is added to the quota for the next time when that thread switches in. This approach maintains the required average $CycPerSwitch_j$ while allowing misses to trigger additional switch points. The same technique, with a different $CycPerSwitch_j$ calculation, was used for $F_{\min/\max}$ enforcement in the work of Gabor et al. [2006, 2007].

Figure 6 shows the achieved conformance level of the different simulation runs, as measured by C_{SLA} . It shows the achieved C_{SLA} of plain SOE (switches only on misses) as well as that of the runs which enforce $F_{\min/\max}=1$ (equalize the speedups of the individual threads [Gabor et al. 2007]) and $C_{SLA}=1$ (maximizes *WeightedSpeedup* under the constraint that the speedups of the two threads are both $\geq 1/2$). The simulations are ordered by the achieved C_{SLA} of plain SOE. 9 out of 38 runs achieve extremely low C_{SLA} , of no more than 0.1, when executed using plain SOE (shown on the left of the figure). This means that one thread lost 95% or more of its performance. In these cases, enforcing SLA or $F_{\min/\max}=1$ improves

the achieved C_{SLA} significantly. Plain SOE achieves high C_{SLA} , of at least 0.9, in 19 out of 38 runs (runs on the right in the figure). In these cases, enforcing $F_{\min}^{\max}=1$ or SLA does not significantly change the achieved C_{SLA} . The results show that SLA enforcement significantly improves the C_{SLA} for the unbalanced cases and has negligible effect on the balanced cases.

Figure 7 shows the achieved *WeightedSpeedup* and aggregate IPC. Figure 7(a) shows that SLA enforcement improves *WeightedSpeedup* by an average of 10% over plain SOE in unbalanced cases (in the cases where $C_{SLA} \leq 0.85$ in plain SOE). The highest *WeightedSpeedup* improvement is of 85% (gzip_025:mgrid_019). When threads are balanced, SLA enforcement has little effect on the *WeightedSpeedup*, with an average of 5% *WeightedSpeedup* improvement over all the runs (balanced and unbalanced). Figure 7(b) shows the comparison of the achieved *WeightedSpeedup* with SLA enforcement and *WeightedSpeedup* with $F_{\min}^{\max}=1$ enforcement. It should be noted that in many cases, $F_{\min}^{\max}=1$ enforcement already achieves most of the *WeightedSpeedup* gains, since it eliminates the extreme cases where one of the thread is significantly slowed down. SLA enforcement still allows for additional *WeightedSpeedup* gains over $F_{\min}^{\max}=1$ enforcement. *WeightedSpeedup* with SLA enforcement is higher in 32 out of 38 runs, and improves by up to 13%. In the other 6 runs we have a small *WeightedSpeedup* drop of up to 0.8%.

As shown in Figure 7(c), there is a drop in the aggregate IPC in the unbalanced cases (shown on the left). In these cases, SLA enforcement biased the execution toward the lower IPC_{no_miss} thread. The aggregate IPC is hardly affected in the balanced runs (runs on the right, which achieve high C_{SLA} in plain SOE). $F_{\min}^{\max}=1$ enforcement usually causes a smaller decrease in the aggregate IPC as compared to SLA enforcement. This is because in all of our runs, the low *CycPerMiss* thread is the low IPC_{no_miss} thread. In these cases, optimizing for *WeightedSpeedup* (under $C_{SLA}=1$ constraint) causes additional biasing toward the lower IPC_{no_miss} (as described in Section 5). Had we optimized for aggregate IPC instead of for *WeightedSpeedup* then we would have minimized the aggregate IPC drop caused by the fairness (or SLA) enforcement, and would have achieved higher aggregate IPC than was achieved by $F_{\min}^{\max}=1$ enforcement (with lower *WeightedSpeedup*).

To gain further insight, we take a closer look at three representative simulation runs. We examine the *WeightedSpeedup* space using the average *CycPerMiss_j* values of both threads as measured in the plain SOE run. The *CycPerMiss* values shown are: *CycPerMiss_{gzip_025}*=14,976; *CycPerMiss_{mgrid_019}*=328; *CycPerMiss_{gap_029}*=1,159; *CycPerMiss_{vpr_008}*=2,268; *CycPerMiss_{perlbnk_024}*=3990. Figure 8 shows the *WeightedSpeedup* space for gzip_025:mgrid_019, gap_029:vpr_008 and perlbnk_024:perlbnk_024. The first pair has the highest improvement in *WeightedSpeedup* when $C_{SLA}=1$ (rather than $F_{\min}^{\max}=1$) is enforced. In the same situation, the second pair has only a slight improvement. Finally, the third pair uses the same application point on both threads.

The 3D graphs in Figure 8 show $F_{\min}^{\max}=1$ as a thin dark stripe. A wider stripe marks $C_{SLA}=1$ and the rest is where $C_{SLA} < 1$. In the two examples

(a) Achieved *WeightedSpeedup*.

(b) Improvement in *WeightedSpeedup* (SLA over $F_{\min}^{\max}=1$ enforcement).

(c) Achieved aggregate IPC.

Fig. 7. Simulation results.

Fig. 8. *WeightedSpeedup* (WS) space of three simulation runs. $CPSw$ is *CycPerSwitch*.

that use different threads (Figures 8(a) and 8(c)), $C_{SLA}=1$ constraint enables us to maximize the *WeightedSpeedup* over that of $F_{\frac{\min}{\max}}=1$. This can be clearly seen in the 2D plots (Figures 8(d) and 8(f)). These 2D plots show the *WeightedSpeedup* : *CycPerSwitch*₂ surface of the 3D plots, for the case of $CycPerSwitch_1=CycPerMiss_1$. In the third case (Figure 8(e)), where both threads have the same *CycPerMiss*, there is little to gain from SLA over $F_{\frac{\min}{\max}}=1$ or plain SOE. In real life, however, the two threads rarely run at exactly the same speed. Instead, they have slightly different *CycPerMiss*'s, which can be translated into *WeightedSpeedup* improvement when using SLA (we calculate fairness parameters every 250,000 cycles, which allows us to closely track performance changes). For the same reason, we see a slight improvement in *WeightedSpeedup* in most of the cases when the same application, from the same starting point, is run in both threads.⁵ As shown, the analytical model correlates with our empirical results and provides a good estimate to the *WeightedSpeedup* gain of the SLA enforcement method over the others. It should be noted that the simulations themselves adjust to the measured *CycPerMiss* every 250,000 cycles (finer granularity than the global average shown in the figure), thus can achieve higher gains than indicated by the analytical model (which uses the average *CycPerMiss* value of the whole run).

⁵We force a starting point offset of 1M instructions when both threads are running the same application point.

7. REDUCING THE POWER UNDER SLA

In this section, we show that maximizing the conformance to SLA (C_{SLA}) can be used as a constraint on one metric while optimizing a different metric. As before, we define SLA using the speedups of the individual threads. The requirement $C_{SLA} = 1$ is satisfied as long as each thread achieves its required speedup. We use $1/2$ as the speedup requirement for two-thread SOE.

An operation mode in which power is reduced while keeping $C_{SLA}=1$ can be useful, for example, when the processor overheats (e.g., limiting the temperature in laptop or handheld devices). In this case, power should be reduced in order to lower the processor temperature. An added advantage is in the case of limited power supplies, where power consumption of the processor should be temporarily reduced due to some circumstances, such as other servers or components temporarily requiring more power than the power supply has been provisioned for.

Power reduction can be done by using Dynamic Voltage Scaling (DVS) [Perring et al. 1998], where voltage and frequency are reduced in order to save power. It can also be done by Dynamic Clock Disabling (DCD), where the clock is dynamically disabled for periods of time, or by throttling resources such as fetch or data cache accesses [Brooks and Martonosi 2001]. The drawback of these approaches is that they reduce the power by reducing the performance of all the threads, regardless of their power consumption. This means that in the case that one thread is a power-hogger, the other thread will have to lose performance as well as in order to reduce the overall power (e.g., processor frequency reduction). We can work around this drawback by selecting the minimal power execution point, out of the acceptable ($C_{SLA}=1$) execution points. In this way, we are biasing execution toward the lower power thread while maintaining SLA. SLA is important in order to avoid starvation of any thread. It should be noted that even after biasing the execution toward the threads with the lower power requirement, we may still have to employ methods such as DVS or DCD in order to guard against pathological cases. Effective biasing will, however, prevent unnecessary performance loss.

The Data Cache Unit (DCU) power is used as a case study for our power reduction technique. We simulated a two-thread SOE processor (Section 9 describes the processor in detail). We measured the DCU power, using access counts multiplied by the energy per access. The energy per access of the simulated data cache (32K, 8-ways, 64 bytes per way, 2 read ports and 1 write port) was estimated using CACTI 4.2 [Tarjan et al. 2006] at 70nm process technology. The average power was periodically calculated for each thread using its access counter, CACTI energy per access, and the cycles it has been running. We then calculate the acceptable ($C_{SLA}=1$) execution points based on the SOE analytical model (see Eq. (11) in Appendix A), and choose the execution point ($CycPerSwitch_1$ and $CycPerSwitch_2$) which is acceptable and which maximizes the execution time of the lower average power per cycle thread. This calculation and thread parameters sampling was done every 250,000 cycles. Using this method, we conceptually bias the execution toward the lower average power thread while maintaining SLA.

Fig. 9. Relative power and achieved C_{SLA} of power reduction execution mode under SLA.

Figure 9 shows the power-reduction execution mode relative to the power of plain SOE as well as the achieved C_{SLA} value of the two execution modes. The simulation runs are ordered by their achieved C_{SLA} of plain SOE. The simulation runs that are balanced in plain SOE, shown on the right side of the figure, are indifferent to the power reduction execution mode. This is because, in these cases, both threads have the same characteristics (many of them are the same application on both threads). Biasing execution toward one of the threads has low potential power gain since both threads are similar (consume the same average power). When threads of different characteristics are executed together, they usually achieve low C_{SLA} , mostly due to the difference in *CycPerMiss* values, as shown in the example in Section 4. In these cases, shown on the left side of the figure, plain SOE achieves low C_{SLA} . Power reduction under SLA execution mode improves the achieved C_{SLA} , usually to a value that is close to 1, and reduces the power by up to 26%. This can also occur when both threads are running the same application.⁶ For example, `ammp_038:ammp_038` (the left-most run) shows 26% power reduction. This occurs due to different phases in the application being executed on the two threads, which allows for power optimization under SLA. The power-reduction execution mode achieved an average of 15% power reduction in the 10 nonbalanced runs, shown on the left side, while maintaining high C_{SLA} .

8. DIFFERENTIATED SERVICE UNDER SLA

SLA can incorporate differentiated service concepts in which each thread may have a different SLA constraint. This could be useful for scheduling threads with different requirements together. For example, one thread may have a real-time requirement, while the other can be a best effort. Even in this case, we may need to guarantee some minimal level of throughput for the best effort thread. Example of such a scenario could be a real-time (online) video encoder thread, that has to process a certain amount of data per second, scheduled together with

⁶We force a starting point offset of 1M instructions in these cases.

a background thread that does some calculation that has less strict throughput requirements. Such a scenario requires slight change to the C_{SLA} metric defined in Eq. (1). It requires a per thread *SLA* constraints, replacing the constant $1/N$ constraint.

SLA-based metric, C'_{SLA} , for differentiated service can be defined as shown in Eq. (10), where C_j is the speedup constraint for thread j , $C_j > 0$. It should be noted that Eq. (1) is a special case of Eq. (10) in which $C_j = 1/N$ for all threads.

$$C'_{SLA} = \begin{cases} \min_j \left(\frac{Speedup_j}{C_j} \right) & \text{if } \min_j \left(\frac{Speedup_j}{C_j} \right) < 1 \\ 1 & \text{otherwise} \end{cases} \quad (10)$$

Using $C_j = 0$ means that thread j is a best-effort thread. This means that the *SLA* constraint allows for that thread not to run at all. In order to allow for $C_j = 0$, Equation (10) needs to be modified to consider $\min_j \frac{Speedup_j}{C_j}$ for all j such that $C_j > 0$ (ignore the best-effort threads).

We have implemented differentiated service support into the simulator, using the model described in Appendix B, where the sum of the speedup constraints is 1 (thread 2 required speedup is 1–first thread speedup). We maintain *CycPerSwitch_j* for each thread such that the *WeightedSpeedup* is maximized under the constraint that $C'_{SLA} = 1$. Appendix B describes the algorithm used for setting the *CycPerSwitch* for both threads as well as the mathematical reasoning behind it.

Figure 10 shows the results of differentiated service simulations for two threads combination: gap_029:vpr_008 and perlbnk_024:perlbnk_024. Each combination was simulated using seven differentiated service constraints, labeled as $\alpha : 1-\alpha$, where α is the constraint on the speedup of the first thread and $1-\alpha$ is the constraint on the speedup of the second thread. The results of the simulation runs are shown in Figures 10(a) and 10(c). Analytical model charts of the same two combinations are shown in Figures 10(b) and 10(d). The analytical model figures show the *WeightedSpeedup* space with annotated points marking the maximum *WeightedSpeedup* point that satisfies the different QoS constraint (each point is labeled by its α). It should be noted that the differentiated service constraints of $\alpha = 50\%$ is the same as the regular *SLA* case (speedup of both threads $\geq 1/2$). Hence, the points marked as 50 in the analytical model Figures 10(b) and 10(d) correspond to the max *WeightedSpeedup* point marked as satisfying *SLA* constraints in Figures 8(c) and 8(e).

Simulation results show that controlling *CycPerSwitch*, as formulated in Appendix B, is effective at maintaining differentiated service constraints. Comparing the results to the analytical model *WeightedSpeedup* charts shows that the trend of *WeightedSpeedup* follows that of the analytical model. In gap_029:vpr_008, the analytical model shows that *WeightedSpeedup* increases with α (*WeightedSpeedup* increase by 0.03), while in the case of perlbnk_024:perlbnk_024, changing α has negligible effect on *WeightedSpeedup* (*WeightedSpeedup* changes by 0.002). This is because in the first case, increasing α increases the time the slower thread gets (the thread with the higher *CycPerMiss*). This means, conceptually, that the faster thread gives up some fraction of the time in favor of the slower thread. The

(a) gap-029:vpr-008 results

(b) gap:vpr analytical model

(c) perlbnk_024:perlbnk_024 results

(d) perlbnk:perlbnk analytical model

Fig. 10. Differentiated service results and analytical model. *WS* is *WeightedSpeedup* and *CPSw* is *CycPerSwitch*.

slower thread gains more speedup than the faster threads loses. In the case of perlbnk_024:perlbnk_024, both threads are from the same application (run at similar speed), hence moving execution cycles from one to the other have smaller effect on the overall *WeightedSpeedup*.

It should be noted that the analytical model projects higher total *WeightedSpeedup* than what we get in the simulation runs. This is due to the inaccuracies of the model, which tends to underestimate the single-threaded performance [Gabor et al. 2006]. This is because sharing of resources reduces the IPC in between misses of SOE, compared to ST. This inaccuracy affect the accuracy of the *WeightedSpeedup* projection, but as our simulation runs show, have small effect on the fairness or C_{SLA} projection (which is used by our SLA enforcement). This can be viewed as the effect of maintaining fairness of multithreaded runs when compared to hypothetical single-thread runs with reduced resources effective size (reduced sizes due to the sharing in the multithreaded runs).

The differentiated SLA demonstrates the effectiveness of using $C'_{SLA}=1$ as a constraint while maximizing the *WeightedSpeedup*. The $C'_{SLA}=1$ constraint defines the range of acceptable execution points ($CycPerSwitch_j$ values), out of which we choose the one that maximizes our target (*WeightedSpeedup*).

9. 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

Table III. Simulated Machine Parameters

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
SOE switch latency	Cycle accurately simulated. Usually takes around 25 cycles.

[Gwennap 1995]. The simulator provides full accurate simulation of the application as well as interrupts, operating system IO and DMA side effects. This simulator and tool chain have been extensively used in many studies [Akkary et al. 2004; Cooksey et al. 2002; Falcon et al. 2004; Gabor et al. 2007; Mutlu et al. 2003; Singhal et al. 2004].

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. Table III summarizes machine configuration parameters. Structure sizes were based on Intel's disclosure of the core processor [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 75ns at 4GHz processor frequency.

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 Reorder Buffer (ROB). 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.⁷ A thread switch causes instructions to be drained from the RS (Reservation Station), ROB and load buffers (LBs). Switch latency, the number of cycles from the start of the switch until the first instruction of the next thread retires, is simulated in a cycle-accurate manner. The switch latency is not constant (e.g., depends 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 approach maintains performance after thread switch events [McNairy and Bhatia 2005]. Some structures, such as the 128 entries 4-way dTLB, include a thread tag per entry, 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 does not forward their data if they are not from the same thread.

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

The misses and cycles counters are sampled every 250,000 cycles, for the recalculation of fairness or SLA parameters. To ensure that all the threads run in every 250,000 cycles, each thread is limited to a maximum of 50,000 cycles before it is forced to switch out.

Spec CPU2000 benchmarks [CPU2000] were simulated on a two-thread SOE configuration. Caches were warmed up using 10,000,000 instructions from each thread (interleaved). 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 state of the fairness or SLA mechanism). When running the same benchmark on both threads, the two threads were offset by 1,000,000 instructions.

We used noncooperative combinations of ST workloads. Equal amounts of work is guaranteed by sampling the simulator separately for each thread when it reaches its designated instruction count. Both threads continue to run, even after one of them is sampled, until the last one reaches its designated instruction count. This guarantees that we use the same work (instruction segment of each thread) in the different runs.

We used 38 combinations of benchmarks. In 18 cases, we used the same benchmark at the same starting point on both threads. Each combination was simulated using plain SOE (no fairness or *SLA* enforcement), with $F_{\frac{\min}{\max}}$ maximization (max-min speedup ratio enforcement [Gabor et al. 2007]) and with *SLA* enforcement while maximizing *WeightedSpeedup* or reducing the power. In addition, we simulated each benchmark separately on the processor in order to obtain its actual 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).

10. RELATED WORK

Fairness has been studied in the context of job scheduling. Job scheduling includes, for example, process scheduling on multiprocessor systems (at the operating system level) and packet stream processing in communication systems (e.g., network packet processing in routers). Job scheduling deals with the decision of how to handle queues of jobs in resource limited environments. Wang and Morris [1985] defined the Q-factor as the mean response time relative to a global first-come-first-served (FCFS) policy. The Q-factor measures both performance and fairness, using FCFS as a baseline. Many communication scheduling algorithms were analyzed by comparing their performance (handling or completion time) to that of General Processor Sharing (GPS), which is a byte-level round-robin. Such algorithms include, for example, Self-Clocked Fair Queuing (SCFQ) [Golestani 1994], Weighted Fair Queuing (WFQ) [Goyal et al. 1997] and Worst-case Fair-Weighted Fair Queuing (WF²Q) [Bennett and Zhang 1996]. These algorithms attempt to create a behavior similar to that of the theoretical GPS model, and hence achieve fairness. Avi-Itzhak and Levy [2004] based their

fairness measure on the service order. Wierman and Harchol-Balter [2003] suggested using $E[T(x)/x]$, the expected normalized response time for job of size (handling time) x , as a measure of fairness. The expected normalized response time is the expected slowdown induced by the scheduling algorithm to a job of size x . Raz et al. [2004] defined a resource-allocation queuing fairness measure (RAQFM). RAQFM is based on the principle that all jobs present in a system at time t deserve an equal share of the resources. Hence, the share of each job at time t is $1/N(t)$, where $N(t)$ is the number of jobs present at time t .

Our $\mathcal{C}_{\text{SLA}}$ metric bares some conceptual resemblance to GPS and RAQFM [Raz et al. 2004]. In both of these job-scheduling concepts, a theoretical system where each job gets an equal share of resources from the time it arrives until it completes, is considered a fair system. These job-scheduling concepts then use this theoretical system in order to determine job scheduling. In our case, we already have threads running, and we need to optimize the processor resources among them. Furthermore, in job-scheduling concepts, the required service time per job is usually not affected by the other coscheduled jobs, while in our case, this is true. In the SOE case, we use the mutual effects between the coscheduled threads in order to optimize the weighted speedup under $\mathcal{C}_{\text{SLA}}=1$ constraints.

Admission control [Guo et al. 2007] has been proposed to reduce fairness issues between concurrent threads by controlling the allowed threads combinations. Admission control uses prestatated or expected demands of jobs to make decisions before the execution. Resource absolute requirements, such as cache or bandwidth requirements, are useful for admission control, since they can be easily summed up. Fairness or SLA enforcement are done among the already running threads and can use the actual execution as a feedback to the enforcement. In this case, relative metrics such as *Speedup* can be effectively used. SLA and fairness enforcement can be used to complement admission control, and their metrics can be used as feedback for future admission control decisions.

Fair cache sharing between multiple coscheduled threads has been shown to be a potential cause of serious problems such as threads starvation. Cache sharing can be extremely unfair when, for example, 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 resource partitioning schemes have been proposed to improve fairness in sharing caches and other resources [Cazorla et al. 2004; Chang and Sohi 2007; Hsu et al. 2006; Iyer 2004; Iyer et al. 2007; Kim et al. 2004; Rafique et al. 2006; Yeh and Reinman 2005]. 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. [Nesbit et al. 2006; 2007] showed that fair memory scheduler can also improve performance.

Fairness of threads was studied in the context of SMT [Cazorla et al. 2006; Cazorla et al. 2004; Luo et al. 2001; Raasch and Reinhardt 1999, 2003; Tullsen and Brown 2001; Chen and Ma 2007]. Tuck and Tullsen [2003] noted that cache and other resource conflicts exist even when SMT processors are designed to minimize conflicts between threads. Raasch and Reinhardt [2003] showed that resource partitioning in SMT improves fairness. For example, statically

partitioned ROB improves fairness as compared to competitive sharing. Luo et al. [2001] used fetch policy as a heuristic to prioritize the different threads, in order to improve fairness. Cazorla et al. [2004, 2006] studied Quality of Service (QoS) enforcement in SMT. They sample the throughput of threads when executed separately (for a short period of time). They then enforce QoS by dynamically managing resources to achieve the operating system QoS requirements (per thread percentage of IPC requirement). Lee and Asanovic [2006] suggested using measurements from various components in order to enforce QoS requirements. Chen and Ma [2007] suggested measuring the single-threaded performance of each thread in an SMT processor while the other thread is being stalled by a cache miss. These measurements can be used to estimate the single-thread IPC of each of the threads, as well as the overall fairness.

11. SUMMARY AND CONCLUSIONS

This article presents a new SLA metric based on requirements rather than on equalizing a measured property or a ratio. Previously suggested fairness metrics, such as $F_{\frac{\min}{\max}}$, $F_{\max-\min}$, and F_{stdev} concentrate on equalizing a parameter (e.g., equalizing the speedups of the individual threads). The use of a fairness metric provides a way to share the hardware resources. A service level agreement is a contract between the software (a thread) and a multithreading or multicore platform. Each thread receives the agreed capacity regardless of the service level provided to other threads. We use SLA as a constraint. In addition, after satisfying SLA, if some processing capacity is left unused as it is often the case, we show that this capacity can be used for optimizing a target metric such as weighted speedup or power.

Our simulation results show that the SLA enforcement mechanism works well for a variety of thread combinations. In some cases, plain SOE achieves high service level even without any enforcement. In these cases, our mechanism has little if any effect on the optimized parameter (*WeightedSpeedup*, aggregate IPC, or power) or on the achieved C_{SLA} . On the other hand, the SLA enforcement mechanism has a significant impact on instances in which plain SOE produces low service level, improving the achieved C_{SLA} as well as the optimized metric. In these cases, we were able to increase *WeightedSpeedup* by an average of 10% or reduce the power by an average of 15%. While these improvements may not be very impressive when considered by themselves, their significance becomes evident when compared with performance losses that may occur by enforcing fairness, if a fairness mechanism is used. Where power reduction is concerned, other methods such as DVS or DCD (discussed in Section 7) may be necessary, in conjunction with SLA, to target tighter power budgets. In these situations, the use of SLA combined with other power reduction methods provides equitable reduction in the service level for various users or applications.

SLA can include priorities between threads by setting a different required value (constraint) to each thread. We have demonstrated such a differentiated service case, showing that SLA approach is useful for setting a constraint on the execution, while allowing for the optimization of a target metric (*WeightedSpeedup*).

For the purpose of introducing the new concepts of SLA and C_{SLA} , SOE is an attractive multithreading approach that allows us to extend an analytical model [Gabor et al. 2006, 2007] rich enough to emulate a processor's shared resources but simple enough to be mathematically tractable. The concept of SLA is not limited however to SOE. For example, it can be used to enforce a service level in SMT and CMP processors (e.g., SLA in shared caches). In order to apply methods such as the one demonstrated here for SOE, one needs to be able to know or be able to estimate how changes in policy will affect the service level and the optimization target (e.g., *WeightedSpeedup* or power as used in our case). In SMT, this can be done using profiling based techniques, where the single-thread IPC of the individual threads is profiled before the threads are executed together. It can also be applied to SMT by sampling each thread by its own at regular intervals [Cazorla et al. 2006] or during cache misses of the other thread [Chen and Ma 2007]. These single-thread IPC samples can then be used, for example, to periodically estimate the fairness and bias the fetch priority of the threads accordingly [Cazorla et al. 2006; Chen and Ma 2007].

The SLA approach can be extended to include multiple metrics. For example, the system may offer SLA in performance (IPC), cache occupancy, bus usage and power budget. In this case, each thread can have four required values, one for each shared resource under SLA (some of which may be 0, for best effort). These per-thread multidimensional SLA requirements will define the set of acceptable execution points, out of which the system will have to choose the one which optimize its target (e.g., *WeightedSpeedup* or power). In some cases, when it is difficult to estimate or measure the effects of the shared resources, a search for the optimal execution point (such that $C_{SLA}=1$) can be performed. In this case the usage of each resource is biased toward some threads, and a feedback loop is required in order to tune the biasing for optimal execution. This sort of multidimensional SLA introduces many challenges. We leave this for future work.

APPENDIXES

APPENDIX A. MAXIMIZING *WEIGHTEDSPEEDUP* UNDER SLA

In order to maximize the weighted speedup (*WeightedSpeedup*) under SLA in two-thread SOE, we have to determine the $CycPerSwitch_j$ that needs to be maintained. Corollary A.1 shows how to obtain the $CycPerSwitch_j$ so that *WeightedSpeedup* is maximized under $C_{SLA}=1$ constraint. The Corollary can be proven algebraically. A high level of proof is given in the analysis in the following text.

The following analysis assumes that $MissLatency > 2SwitchLatency > 0$ and that, without loss of generality, $MissLatency - 2SwitchLatency \leq CycPerMiss_1 \leq CycPerMiss_2$. It also uses $MissLatency - 2SwitchLatency \leq CycPerSwitch_j \leq CycPerMiss_j$ for $j=1$ and 2. $CycPerSwitch_j$ should not be lower than $MissLatency - 2SwitchLatency$, otherwise threads might switch back after a miss, before that miss has been resolved (this is a reasonable assumption according to our simulations). $CycPerSwitch_j$ cannot be more than $CycPerMiss_j$ as there is no way to force the average number of cycles between switches to be

more than those between misses (we switch on misses as well). Based on these assumptions and on the SOE analytical model (see Eq. (5) and Eq. (7)), we can make the following observations (all of which can be proved):

- (1) *WeightedSpeedup* is monotonically nondecreasing with respect to $CycPerSwitch_1$. Hence, *WeightedSpeedup* is maximized with respect to $CycPerSwitch_1$ when $CycPerSwitch_1 = CycPerMiss_1$.
- (2) *WeightedSpeedup* is monotonically nondecreasing or nonincreasing with respect to $CycPerSwitch_2$ (there is no local maximum in the range of $MissLatency - 2SwitchLatency \leq CycPerSwitch_2 \leq CycPerMiss_2$). Hence, *WeightedSpeedup* is maximized with respect to $CycPerSwitch_2$ when $CycPerSwitch_2 = MissLatency - 2SwitchLatency$ or when $CycPerSwitch_2 = CycPerMiss_2$.
- (3) *WeightedSpeedup* is maximized when $CycPerSwitch_1 = CycPerMiss_1$ and $CycPerSwitch_2 = MissLatency - 2SwitchLatency$ or $CycPerSwitch_2 = CycPerMiss_2$ (based on observations 1 and 2).
- (4) Assuming that $CycPerSwitch_1 = CycPerMiss_1$ (which maximizes *WeightedSpeedup* with respect to $CycPerSwitch_1$ per observation 1), *WeightedSpeedup* is nondecreasing with $CycPerSwitch_2$ when: $CycPerMiss_2 \leq CycPerMiss_1 + 2SwitchLatency$ or $CycPerMiss_2 > CycPerMiss_1 + 2SwitchLatency$ && $\frac{2CycPerMiss_2SwitchLatency}{CycPerMiss_2 - CycPerMiss_1 - 2SwitchLatency} \geq MissLatency$.

In order to maximize *WeightedSpeedup* under SLA constraint, we need to define a range of $CycPerSwitch_2$ in which $C_{SLA}=1$ when $CycPerSwitch_1 = CycPerMiss_1$ (based on observations 1 and 2). This means maximizing *WeightedSpeedup* with respect to $CycPerSwitch_2$ under the constraints of $Speedup_1 \geq 1/2$, $Speedup_2 \geq 1/2$ and $CycPerSwitch_1 = CycPerMiss_1$. This yields Eq. (11) which defines the range of $CycPerSwitch_2$ in which QoS constraints are satisfied.

$$\frac{CycPerMiss_2(CycPerMiss_1 + 2SwitchLatency)}{CycPerMiss_2 + 2MissLatency} \leq CycPerSwitch_2 \leq \frac{CycPerMiss_1 + 2MissLatency - 2SwitchLatency}{1} \quad (11)$$

Eq. (11) defines the range where SLA is obtained ($C_{SLA}=1$). According to observation 2, *WeightedSpeedup* is maximized under $C_{SLA}=1$ constraint on one of this range edges (or on both).⁸ Observation 4 determines the edge. This leads to the following corollary, which defines how to fairly maximize *WeightedSpeedup* in a two-thread SOE.

COROLLARY A.1. *Assuming that $MissLatency > 2SwitchLatency > 0$ and that, without loss of generality, $(MissLatency - 2SwitchLatency) < CycPerMiss_1 \leq CycPerMiss_2$, then *WeightedSpeedup* is maximized under $C_{SLA}=1$ constraint*

⁸Figure 5(b) of the example in Section 5 shows a case where maximum *WeightedSpeedup* is found when $CycPerSwitch_2$ is minimized (*Cond* in Corollary A.1 is false). Figure 8(e) is an example of the opposite case, where maximum *WeightedSpeedup* is found when $CycPerSwitch_2$ is maximized (*Cond* in Corollary A.1 is true).

when:

$$\begin{aligned} CycPerSwitch_1 &= CycPerMiss_1 \\ CycPerSwitch_2 &= \begin{cases} T_1 & \text{when } Cond \text{ is true} \\ T_2 & \text{otherwise} \end{cases} \end{aligned}$$

Where:

$$\begin{aligned} T_1 &= CycPerMiss_1 + 2MissLatency - 2SwitchLatency \\ T_2 &= \frac{CycPerMiss_2(CycPerMiss_1 + 2SwitchLatency)}{CycPerMiss_2 + 2MissLatency} \\ Cond &: CycPerMiss_2 \leq CycPerMiss_1 + 2SwitchLatency \parallel \\ &\quad \left(CycPerMiss_2 > CycPerMiss_1 + 2SwitchLatency \ \&\& \right. \\ &\quad \left. \frac{2CycPerMiss_2SwitchLatency}{CycPerMiss_2 - CycPerMiss_1 - 2SwitchLatency} \right. \\ &\quad \left. \geq MissLatency \right) \end{aligned}$$

We use Corollary A.1 to set $CycPerSwitch_j$ in the empirical studies of Section 6. We also set $CycPerSwitch_j$ to be at least $MissLatency - 2SwitchLatency$ to ensure that misses are resolved by the time we perform switches.

APPENDIX B. SOE AND DIFFERENTIATED SERVICE

Differentiated service is when each thread has a different SLA constraint (Section 8). The same method shown in Appendix A can be used to analyze the differentiated service case. This case, however, is more complex, as it is not guaranteed that there exists a point where $CycPerSwitch_1 = CycPerMiss_1$ which satisfies the differentiated SLA constraints. It can be shown that, if any point exists that satisfied these SLA constraints, then there is also at least one point on the border of $CycPerSwitch_2 = CycPerMiss_2$ or $CycPerSwitch_1 = CycPerMiss_1$ that satisfies these constraints. The rest of the appendix discusses the differentiated service case for a two-thread SOE model. Empirical results of using this model are shown in Section 8.

The following discussion uses C'_{SLA} , the differentiated service version of C_{SLA} as shown in Eq. (10). Let us use α as the SLA constraint for the first thread and $1-\alpha$ for the second thread, where $0 < \alpha < 1$. In this case, the range where $C'_{SLA}=1$ and $WeightedSpeedup$ is maximized is usually found where $CycPerSwitch_1 = CycPerMiss_1$ and $CycPerSwitch_2$ is in the range defined in Eq. (12).

$$\frac{CycPerMiss_2(CycPerMiss_1 + 2SwitchLatency)(1 - \alpha)}{\alpha \cdot CycPerMiss_2 + MissLatency} \leq CycPerSwitch_2 \leq \frac{CycPerMiss_1(1 - \alpha) + MissLatency - 2SwitchLatency}{\alpha} \quad (12)$$

For small α values, when $0 < \alpha < \frac{CycPerMiss_1 - MissLatency + 2SwitchLatency}{CycPerMiss_1 + CycPerMiss_2 + 2SwitchLatency}$, the solution for maximizing $WeightedSpeedup$ is on the other border of $CycPerSwitch_2 =$

$CycPerMiss_2$ and $CycPerSwitch_1$ is in the range defined by Eq. (13).

$$\frac{CycPerMiss_1(CycPerMiss_2 + 2SwitchLatency)\alpha}{CycPerMiss_1(1 - \alpha) + MissLatency} \leq CycPerSwitch_1 \leq \frac{MissLatency + 2SwitchLatency(\alpha - 1) + \alpha \cdot CycPerMiss_2}{\alpha - 1} \quad (13)$$

Our domain of $0 < 2SwitchLatency < MissLatency$ and $MissLatency - 2SwitchLatency \leq CycPerMiss_1 \leq CycPerMiss_2$ guarantees that there is always some point that will satisfy the differentiated SLA constraints given as α and $1 - \alpha$. It should be noted that Eq. (11) is a special case of 12, in which $\alpha = 1/2$.

The empirical results in Section 8 are for the described model, where the sum of the speedup constraints is 1 (thread 2 required speedup is $1 -$ first thread speedup). In order to determine the $CycPerSwitch$ for both threads we used Eq. (12), with *cond* from Corollary A.1 determining which limit to use for $CycPerSwitch_2$ (in order to maximize *WeightedSpeedup*). When the range defined by Equation (12) was not achievable (greater than $CycPerMiss_2$), we used Equation (13) to set $CycPerSwitch_1$ (using maximum $CycPerSwitch_1$ that is in that range).

It is also possible to analyze the general two-thread differentiated service case. In this case, the speedup of the first thread must be at least C_1 , and the second thread speedup must be at least C_2 , $C_1 > 0$ and $C_2 > 0$. The analysis follows the same approach as shown earlier, yielding the domain in which the given SLA constraints can be met and how to maximize *WeightedSpeedup* in those cases. The analysis of this case has an additional range in which the SLA constraints cannot be met (in which case we may wish to maximize C'_{SLA}). The analysis of the general case is not in the scope of this article.

REFERENCES

- AKKARY, H., SRINIVASAN, S. T., KOLTUR, R., PATIL, Y., AND REFAAI, W. 2004. Perceptron-based branch confidence estimation. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA-10)*. IEEE, 265.
- AVI-ITZHAK, B. AND LEVY, H. 2004. On measuring fairness in queues. *Adv. Appl. Probab.* 36, 3, 919–936.
- BENNETT, J. AND ZHANG, H. 1996. WF2Q: worst-case fair weighted fair queueing. In *Proceedings of the 15th Annual Joint Conference of the IEEE Computer Societies*. IEEE, 1.
- BORKENHAGEN, J. M., EICKEMEYER, R. J., KALLA, R. N., AND KUNKEL, S. R. 2000. A multi-threaded PowerPC processor for commercial servers. *IBM J. Res. Dev.* 44, 6, 885–898.
- BROOKS, D. AND MARTONOSI, M. 2001. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA-7)*. IEEE, 171–182.
- CAZORLA, F., FERNANDEZ, E., RAMIREZ, A., AND VALERO, M. 2003. Improving memory latency aware fetch policies for SMT processors. In *Proceedings of the 5th International Symposium on High-Performance Computing*. Springer.
- CAZORLA, F., KNJNENBURG, P., SAKELLARIOU, R., FERNANDEZ, E., AND RAMIREZ, A. 2006. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Trans. Comput.* 55, 7, 785.
- CAZORLA, F., RAMIREZ, A., VALERO, M., KNJNENBURG, P., SAKELLARIOU, R., AND FERNANDEZ, E. 2004. QoS for high-performance SMT processors in embedded systems. *Micro. IEEE* 24, 4, 24–31.

- CAZORLA, F. J., KNJNENBURG, P. M., SAKELLARIOU, R., FERNANDEZ, E., RAMIREZ, A., AND VALERO, M. 2004. Feasibility of QoS for SMT by resource allocation. In *Proceedings of the International Euro-Par Conference*. Springer, 535–540.
- CAZORLA, F. J., RAMIREZ, A., VALERO, M., AND FERNANDEZ, E. 2004. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-37)*. ACM, 171–182.
- CHANG, J. AND SOHI, G. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Super-computing*. ACM, 242–252.
- CHEN, S. AND MA, P. 2007. FROCM: A fair and low-overhead method in SMT processor. In *Proceedings of the 3rd International Conference on High-Performance Computing and Communications*. Springer.
- 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.
- 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 *Proceedings of the International Symposium on High-Performance Computing (ISHPC '97)*. Springer, 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 *Proceedings of ISCA-18*. 362–369.
- GABOR, R., WEISS, S., AND MENDELSON, A. 2006. Fairness and throughput in switch on event multi-threading. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-39)*. ACM, 149–160.
- GABOR, R., WEISS, S., AND MENDELSON, A. 2007. Fairness enforcement in switch on event multi-threading. *ACM Trans. Archit. Code Optim.* 4, 3, 15.
- GOLESTANI, S. 1994. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of the 13th Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM'94)*. IEEE, 636–646.
- GOYAL, P., LAM, S., AND VIN, H. 1997. Determining end-to-end delay bounds in heterogeneous networks. *Multimedia Syst.* 5, 3, 157–163.
- GUO, F., SOLIHIN, Y., ZHAO, L., AND IYER, R. 2007. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-40)*. ACM, 343–355.
- GWENNAP, L. 1995. Intel's P6 uses decoupled superscalar design. *Micro-processor Rep.* 9, 2.
- HALFHILL, T. R. 2006. Intel goes quad. *Micro-processor Rep.*
- HSU, L., REINHARDT, S., IYER, R., AND MAKINENI, S. 2006. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT-15)*. ACM, 13–22.
- IYER, R. 2004. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th Annual International Conference on Super-Computing (ICS '04)*. ACM, 257–266.
- IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 25–36.
- JAIN, R., CHIU, D., AND HAWES, W. 1998. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Arxiv preprint cs.NI/9809099. <http://adsabs.harvard.edu/abs/1998cs.....9099J>
- KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip Multi-processor architecture. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT-13)*. ACM, 111–122.
- KONDO, M., SASAKI, H., AND NAKAMURA, H. 2007. Improving fairness, throughput and energy-efficiency on a chip multi-processor through DVFS. *SIGARCH Comput. Archit. News* 35, 1, 31–38.

- KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. 2005. Niagara: A 32-way multi-threaded Sparc processor. *IEEE Micro* 25, 2, 21–29.
- KREWELL, K. 2006. Intel looks to core for success. Micro-processor Rep.
- LEE, J. AND ASANOVIC, K. 2006. METERG: Measurement-based end-to-end performance estimation technique in QoS-capable multiprocessors. In *Proceedings of the 12th IEEE Real-time and Embedded Technology and Applications Symposium*. IEEE, 135–147.
- LUO, K., GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, 164–171.
- MCGREGOR, J. 2007. The New x86 Landscape. Micro-processor rep.
- McNAIRY, C. AND BHATIA, R. 2005. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro* 25, 2, 10–20.
- 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 *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA-9)*. IEEE, 129.
- NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. 2006. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-39)*. ACM, 208–222.
- NESBIT, K. J., LAUDON, J., AND SMITH, J. E. 2007. Virtual private caches. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ICSA-34)*. 57–68.
- PAWLOWSKI, S. 2007. Petascale computing research challenges: A many core perspective. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA 2007)*. IEEE, 96.
- PERING, T., BURD, T., AND BRODERSEN, R. 1998. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'98)*. IEEE, 76–81.
- RAASCH, S. E. AND REINHARDT, S. K. 1999. Applications of thread prioritization in SMT processors. In *Proceedings of the Workshop on Multi-threaded Execution and Compilation*. ACM.
- RAASCH, S. E. AND REINHARDT, S. K. 2003. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architecture and Compilation Techniques (PACT-12)*. ACM, 15–25.
- RAFIQUE, N., LIM, W.-T., AND THOTTETHODI, M. 2006. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT-15)*. ACM, 2–12.
- RAZ, D., LEVY, H., AND AVI-ITZHAK, B. 2004. A resource-allocation queueing fairness measure. *ACM SIGMETRICS Perform. Eval. Rev.* 32, 1, 130–141.
- ROSENBLUM, M. AND GARFINKEL, T. 2005. Virtual machine monitors: Current technology and future trends. *Computer* 38, 5, 39–47.
- 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 Tech. J.* 8.
- SNAVELY, A. AND TULLSEN, D. M. 2000. Symbiotic job-scheduling for a simultaneous multi-threading processor. *SIGPLAN Not.* 35, 11, 234–244.
- TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. 2006. CACTI 4.0. Tech. rep. HP Laboratories.
- THEKKATH, R. AND EGGERS, S. J. 1994. The effectiveness of multiple hardware contexts. *SIGPLAN Not.* 29, 11, 328–337.
- TUCK, N. AND TULLSEN, D. M. 2003. Initial observations of the simultaneous multi-threading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architecture and Compilation Techniques (PACT-12)*. ACM, 26.
- TULLSEN, D. M. AND BROWN, J. 2001. Handling long-latency loads in a simultaneous multi-threading processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-34)*. ACM, 318–327.
- WANG, Y. AND MORRIS, R. 1985. Load sharing in distributed systems. *IEEE Trans. Comput.* 100, 34, 204–217.

- WIERMAN, A. AND HARCHOL-BALTER, M. 2003. Classifying scheduling policies with respect to unfairness in an M/GI/1. *ACM SIGMETRICS Perform. Eval. Rev.* 31, 1, 238–249.
- YEH, T. AND REINMAN, G. 2005. Fast and fair: Data-stream quality of service. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 237–248.

Received April 2008; revised October 2008; accepted February 2009