Per-Thread Cycle Accounting in SMT Processors

Stijn Eyerman Lieven Eeckhout
ELIS Department, Ghent University, Belgium
Email: {seyerman,leeckhou}@elis.UGent.be

Abstract

This paper proposes a cycle accounting architecture for Simultaneous Multithreading (SMT) processors that estimates the execution times for each of the threads had they been executed alone, while they are running simultaneously on the SMT processor. This is done by accounting each cycle to either a base, miss event or waiting cycle component during multi-threaded execution. Single-threaded alone execution time is then estimated as the sum of the base and miss event components; the waiting cycle component represents the lost cycle count due to SMT execution. The cycle accounting architecture incurs reasonable hardware cost (around 1KB of storage) and estimates single-threaded performance with average prediction errors around 7.2% for two-program workloads and 11.7% for four-program workloads.

The cycle accounting architecture has several important applications to system software and its interaction with SMT hardware. For one, the estimated single-thread alone execution time provides an accurate picture to system software of the actually consumed processor cycles per thread. The alone execution time instead of the total execution time (timeslice) may make system software scheduling policies more effective. Second, a new class of thread-progress aware SMT fetch policies based on per-thread progress indicators enable system software level priorities to be enforced at the hardware level.

Categories and Subject Descriptors C.1.4 [*Processor architectures*]: Parallel architectures

General Terms Design, Experimentation, Performance

Keywords Simultaneous Multithreading (SMT), Cycle accounting, Thread-progress aware fetch policy

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

1. Introduction

Simultaneous Multithreading (SMT) processors [28, 29] seek at improving single-core processor utilization by sharing resources across multiple active threads. Whereas resource sharing increases overall system throughput, it may affect single-thread performance in unpredictable ways, and may even starve threads. However, system software, e.g., the operating system (OS) or virtual machine monitor (VMM), assumes that all threads make equal progress when assigning timeslices. The reason for this simplistic assumption is that there is no mechanism for tracking per-thread progress during SMT execution.

This paper proposes a cycle accounting architecture for SMT processors for at run time per-thread performance accounting. For threads executing on the SMT processor, the counter architecture accounts each cycle into three cycle components: (i) base cycles: the processor consumes cycles doing computation work for the given thread, (ii) miss event cycles: the processor consumes cycles handling miss events for the given thread (cache misses, TLB misses and branch mispredictions), and (iii) waiting cycles: the processor is consuming cycles for another thread and can therefore not make progress for the given thread. Computing these three cycle components at run time for each of the co-executing threads provides insight into how single-thread performance is affected by SMT execution, e.g., a thread starving because of another thread clogging resources will experience a large number of waiting cycles. The cycle accounting architecture proposed in this paper predicts per-thread cycle components within at most 6% compared to the cycle components computed under single-threaded execution. A thread's *alone execution time*, i.e., the execution time had the thread been executed alone on the processor, is estimated by adding the base cycle component to the miss event cycle components. The cycle accounting architecture is shown to be accurate across fetch policies with average errors around 7.2% for two-program workloads and 11.7% for four-program workloads for predicting a thread's alone execution time. The counter architecture incurs a reasonable hardware cost around 1KB of storage.

There are several applications for the cycle accounting architecture in support of system software running on SMT hardware. First, communicating the per-thread alone execu-

tion time to system software instead of the total execution time (timeslice) provides a more accurate measurement of the per-thread consumed processor cycles, and may make system software scheduling policies more effective; both classical OS scheduling policies as well as SMT hardware aware policies such as symbiotic job scheduling [23, 24] and non-work-conserving policies [13] may benefit from knowing per-thread alone execution times. Also, the alone execution time is a useful metric when selling compute cycles in a data center consisting of SMT processors, i.e., customers get an accurate bill for the consumed compute cycles. Second, per-thread cycle accounting eliminates the single-thread sampling phase (and its overhead) in several SMT optimizations, such as symbiotic job scheduling with priorities [24], hill-climbing SMT resource partitioning [5], and quality-of-service (QoS) on SMT processors [2]. Third, per-thread progress which is defined as the ratio of the per-thread alone execution time to the total execution time, enables a new class of SMT fetch policies based on per-thread progress indicators. These thread-progress aware SMT fetch policies enforce system software priorities in hardware, i.e., they enforce one thread to make faster single-thread progress than another thread proportionally to the difference in system-level priorities. Existing SMT fetch policies on the other hand do not control per-thread progress. The abstraction provided to system software, consumed processor cycles or alone execution time, is consistent with the notion of a timeslice used by system software for managing QoS, priorities and performance isolation through time multiplexing [25].

2. Interval Analysis: Analyzing Miss Events

Before describing how miss events affect the cycle components under SMT execution, we first describe how miss events affect performance on a single-threaded out-of-order processor. For doing so, we build on the prior analytical performance modeling work by Karkhanis and Smith [17] and Eyerman et al. [12], called *interval analysis*. With interval analysis, execution time is partitioned into discrete intervals by disruptive miss events such as cache misses, TLB misses and branch mispredictions. The basis for the model is that an out-of-order processor is designed to stream instructions through its various pipelines and functional units; and, under optimal conditions (no miss events), a well-balanced design sustains a level of performance more-or-less equal to its pipeline front-end *dispatch* width — we refer to dispatch as the point of entering the instructions from the front-end pipeline into the reorder buffer and issue queues.

The interval behavior is illustrated in Figure 1 which shows the number of dispatched instructions on the vertical axis versus time on the horizontal axis. By dividing execution time into intervals, one can analyze the performance behavior of the intervals individually.

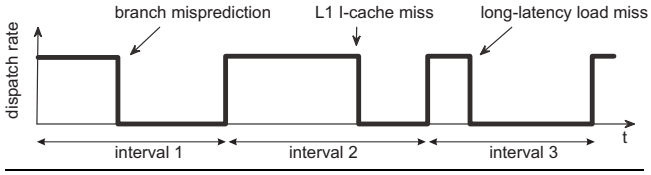

Figure 1. Interval analysis analyzes on an interval basis by disruptive miss events.

In particular, one can, based on the type of interval (the miss event that terminates it), describe and determine the performance penalty per miss event:

- For an *I-cache miss (or I-TLB miss)*, the penalty equals the miss delay, i.e., the time to access the next level in the memory hierarchy.
- For a *branch misprediction*, the penalty equals the time between the mispredicted branch being dispatched and new instructions along the correct control flow path being dispatched. This delay includes the branch resolution time plus the front-end pipeline depth [12].
- Upon a *long-latency load miss*, i.e., a last-level (L2 or L3) D-cache load miss or a D-TLB load miss, the processor back-end will stall because of the reorder buffer (ROB), issue queue, or rename registers getting exhausted. As a result, dispatch will stall. When the miss returns from memory, instructions at the ROB head will be committed, and new instructions will enter the ROB. The penalty for a long-latency D-cache miss thus equals the time between dispatch stalling upon a full ROB and the miss returning from memory. In case multiple independent long-latency load misses make it into the ROB simultaneously, both will overlap their execution, thereby exposing memory-level parallelism (MLP) [6], provided that a sufficient number of outstanding long-latency loads are supported by the hardware.
- Chains of dependent instructions, L1 data cache misses and long-latency functional unit instructions (divide, multiply, etc.), or store instructions, may cause a resource (e.g., reorder buffer, issue queue, physical register file, write buffer, etc.) to fill up. A *resource stall* as a result of it may (eventually) stall dispatch. The number of cycles where dispatch stalls due to a resource stall are attributed to the instruction at the ROB head, i.e., the instruction blocking commit and thereby stalling dispatch.

Eyerman et al. [12] describe how these insights can lead to a counter architecture to be implemented in hardware to compute cycle stacks on an out-of-order processor. A cycle stack consists of a base cycle component plus a number of cycle components that reflect 'lost' cycle opportunities due to miss events such as branch mispredictions, and cache and TLB misses. The breakdown of the total execution time into components is often referred to as a cycle 'stack' because the cycle components are placed one on top of another with

the base cycle component being shown at the bottom of the bar.

3. Per-Thread SMT Cycle Stacks

3.1 Overall goal

The goal of per-thread cycle accounting is to estimate per-thread *alone execution times* during SMT execution. To do so, we estimate the following three *cycle components* for each thread:

- **Base cycle component.** The processor consumes cycles doing computation work.
- **Miss event cycle component.** The processor consumes cycles handling miss events such as cache misses, TLB misses, and branch mispredictions.
- **Waiting cycle component.** The processor consumes cycles for another thread, and therefore cannot make progress for the given thread.

The *total execution time* in SMT execution is the total number of execution cycles, i.e., the sum of base, miss event and waiting cycle components. The per-thread *alone execution time* is estimated as the sum of the base and miss event cycle components, i.e., this is the thread's (estimated) total execution time if the thread would be executed in single-threaded mode. The waiting cycle component thus quantifies by how much the given thread gets slowed down because of SMT execution.

Per-thread cycle accounting estimates these cycle components based on a number of counts that are computed at run time. In particular, cycle accounting uses the notion of a *dispatch slot* (i.e., a 4-wide dispatch processor has 4 *dispatch slots* per cycle) and counts the number of per-thread base, miss event and waiting dispatch slots. The accounting architecture also estimates the increase in the number of per-thread miss events (cache and TLB misses, and branch mispredictions) due to sharing during SMT execution. Finally, it also computes the decrease in per-thread MLP due to SMT execution — single-threaded execution allocates all ROB resources and therefore exposes more per-thread MLP than SMT execution.

We can estimate the cycle components based on these counts. The base cycle component is the number of dispatch slots divided by the processor dispatch width. The miss event cycle components are computed as follows: we first divide the number of miss event dispatch slots by the processor dispatch width, and subsequently divide by the (estimated) increase in the number of per-thread misses due to multithreading — this is referred to as *sharing miss correction*. For the long-latency load miss event slots, we also divide by the reduction in per-thread MLP due to multithreading — this is called *MLP correction*. The waiting cycle component is estimated as the total SMT cycle count minus the base cycle component and the miss event cycle components.

We now discuss how the per-thread cycle accounting architecture computes these counts. This is done in a number of steps. We first discuss the case without miss events, then we discuss the different types of miss events and make a distinction between front-end and back-end misses, and we discuss how we estimate the number of sharing misses. Finally, we discuss the hardware cost.

3.2 No miss events

Figure 2(a) shows the progress for two threads co-executing on an SMT processor in the absence of miss events. We assume a two-thread processor and a round-robin fetch policy for the purpose of illustration; this does not affect the generality of the formulation though, i.e., the description can be trivially extended to more than two threads and other fetch policies (as we will demonstrate later in this paper). Each square in the figure represents a dispatch slot. In cycle x, 4 instructions are dispatched from thread A; these slots are counted as *base* dispatch slots. Thread B on the other hand, cannot make progress; thread B therefore gets 4 *waiting* dispatch slots accounted. In cycle $x + 1$, thread B gets 4 base slots and thread A gets 4 waiting slots, etc. Per-thread performance halves compared to single-thread performance because of SMT execution in the absence of miss events.

3.3 Front-end miss events

We identify two types of front-end miss events: instruction cache and TLB misses, and branch mispredictions.

3.3.1 Instruction cache misses

In the event of an L1/L2 I-cache miss or I-TLB miss, as is the case for thread A in Figure 2(b), the processor will no longer be able to dispatch instructions for thread A during a number of cycles equal to the miss delay. The miss delay equals the access time to the next level in the memory hierarchy, and the penalty slots are to be accounted as *miss event* dispatch slots. The other thread, thread B in the example, will be able to dispatch instructions into the pipeline at a rate of 4 instructions per clock cycle. As a result, thread B benefits from the I-cache miss in thread A, i.e., there are no more waiting dispatch slots for thread B under the I-cache miss of thread A — this is where the benefit of SMT execution comes from: thread B can dispatch instructions while the I-cache miss is being resolved for thread A.

3.3.2 Branch mispredictions

Figure 2(c) illustrates what happens upon a branch misprediction for thread A. When thread A dispatches a mispredicted branch, instructions along the incorrect control flow path will enter the processor back-end until the branch gets resolved and new instructions along the correct control flow path enter the pipeline. The penalty for the mispredicted branch equals the time between the mispredicted branch being dispatched and correct path instructions being dispatched after the branch resolution: these dispatch slots are counted

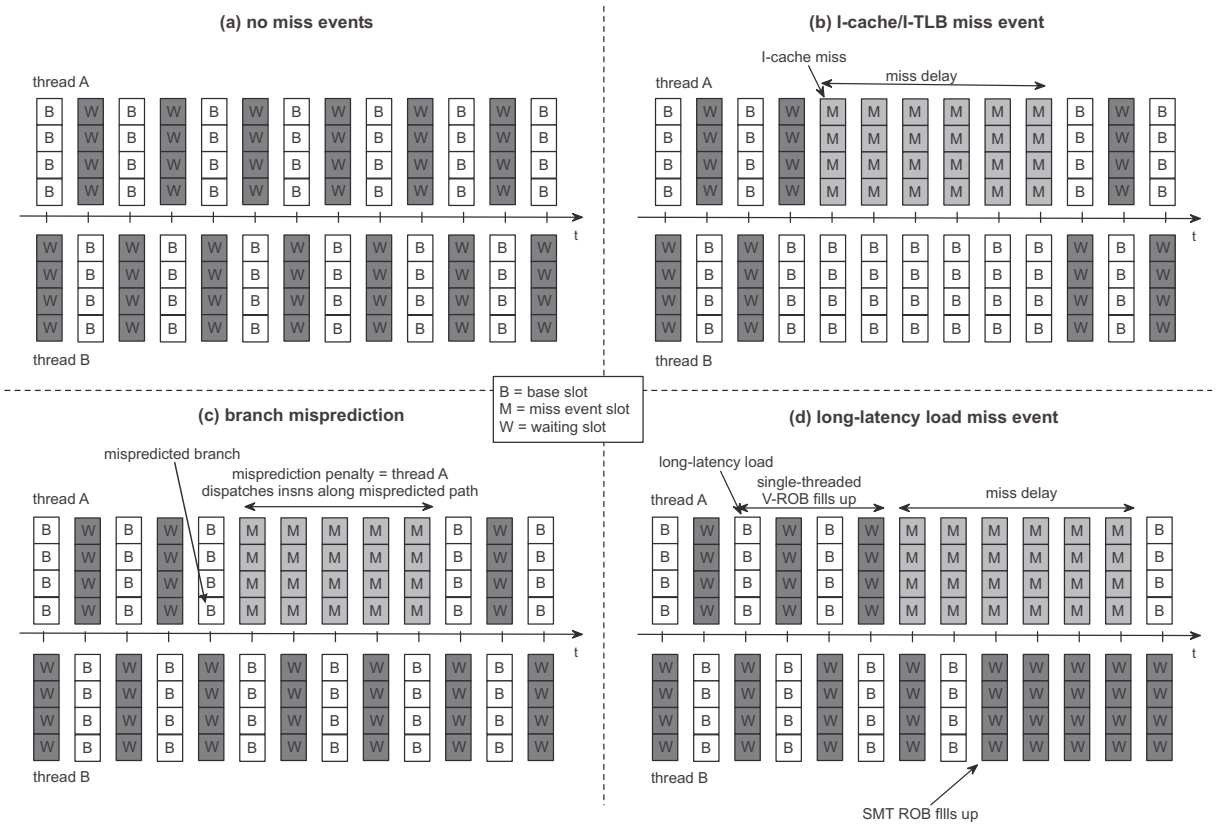

Figure 2. SMT processor execution (a) in the absence of miss events, (b) in the presence of an I-cache/I-TLB miss event, (c) branch misprediction, and (d) long-latency load miss.

as *miss event* slots. These miss event slots include the branch resolution time (which is determined by the critical path of instructions leading to the mispredicted branch) plus the front-end pipeline depth. Underneath the penalty for a mispredicted branch, the other thread(s) will continue dispatching instructions; i.e., the performance of thread B is unaffected by the mispredicted branch for thread A.

3.3.3 Implementation

Accurately counting front-end miss event dispatch slots requires that (i) we do not count I-cache and I-TLB misses along mispredicted paths, (ii) we count the miss delay for I-cache and I-TLB misses, and (iii) we count the number of cycles between dispatching the mispredicted branch and dispatching instructions along the correct path once the mispredicted branch is resolved. We therefore use a front-end miss event table (FMT), inspired by the design by Eyerman et al. [12], which is a circular buffer with as many rows as the processor supports outstanding branches; there is one FMT per thread. We also have a timestamp counter that is incremented upon each dispatch slot. A row is allocated in the FMT per outstanding branch, and counts the number of base, waiting, and miss event (L1 I-cache, L2 I-cache, and I-TLB) slots up to the next branch. When dispatching an instruction, we increment the base dispatch slot counter; for a branch instruction, we also record the current timestamp. In case an instruction is dispatched for another thread, we increment the waiting dispatch slot counter in the absence of a miss event, and increment the miss event dispatch slot counter in case of a miss event. When committing a branch, the counters of the associated row are added to the respective global dispatch slot counters, and the row is de-allocated. When resolving a mispredicted branch, the difference between the recorded and current timestamp is added to the global branch misprediction miss event slot counter — this is the number of dispatch slots since the mispredicted branch was dispatched — and all dispatch slots hereafter are counted as miss event slots until correct-path instructions are being dispatched; in addition, the subsequent rows are de-allocated.

3.4 Back-end miss events

3.4.1 Long-latency load misses

The situation gets more complicated when estimating the penalty due to long-latency loads. Recall that the penalty for a long-latency load under single-threaded execution is the time between the ROB filling up and the miss returning from memory. The key problem now is to estimate when the ROB would fill up under single-threaded execution, during the SMT execution. We do this by keeping track of all the base and waiting dispatch slots since the oldest instruction

in the reorder buffer for the given thread. The point in time where the number of 'in-flight' base and waiting slots, called the single-thread *virtual ROB (V-ROB)*, equals the SMT processor's ROB size, signals the point where the ROB would get exhausted under single-threaded execution. This is illustrated in the example given in Figure 2(d) in which the 16-entry ROB would get exhausted by thread A under single-threaded execution after four cycles — dispatching 16 instructions at a dispatch rate of 4 takes 4 cycles, or 8 base slots plus 8 waiting slots. Once this point of V-ROB exhaustion is reached, we start counting miss event slots, i.e., long-latency load (L2, L3, or D-TLB) miss event dispatch slots; and for each dispatch slot from this point on where a useful instruction is dispatched, we increment the number of base dispatch slots and decrement the number of waiting dispatch slots. When eventually the processor stalls because of resource clogging, the other thread(s) are accounted waiting slots, see the example given in Figure 2(d).

We also measure the amount of per-thread MLP under SMT execution; this is done by counting the number of outstanding long-latency load misses if at least one is outstanding. In addition, we also estimate the amount of MLP under single-threaded execution (see Section 3.4.4 for how this is done). The ratio of the MLP under SMT execution divided by the MLP under single-threaded MLP quantifies the decrease in per-thread due to SMT execution, which is used for the MLP correction.

3.4.2 Other resource stalls

Other resource stalls due to long-latency functional units (divide, multiply, etc.), L1 data cache misses, store buffer stalls, etc. are accounted as miss event slots following the approach for the long-latency loads, i.e., a miss event slot is accounted in case a thread's V-ROB size equals the processor's ROB size.

3.4.3 Stall and flush fetch policies

Some SMT fetch policies, such as those proposed in [10, 27], stall and/or flush threads to prevent threads from clogging resources due to long-latency loads. In case a thread is stalled by the fetch policy, subsequent dispatch slots are counted as waiting dispatch slots (and when the V-ROB size equals the machine's ROB size, miss event slots are accounted — this is the point in time where single-threaded execution would fill up the ROB). In case a thread is (partially) flushed, the number of flushed instructions is subtracted from the base dispatch slot counter and added to the waiting dispatch slot counter.

3.4.4 Implementation

For computing the back-end miss event cycle components, we need three additional hardware structures.

First, we keep track of the per-thread V-ROB occupancy during single-threaded execution using a *V-ROB Occupancy Counter (VOC)*. The VOC counts the number of per-thread in-flight base and waiting dispatch slots. In the absence of a miss event for the given thread, the VOC is incremented upon instruction dispatch from whatever thread — single-threaded execution would dispatch one instruction per dispatch slot. For each instruction that is dispatched we keep track of the number of waiting slots since the prior dispatched instruction; we keep track of these associated waiting slots in the reorder buffer. Upon instruction commit, we subtract the number of associated waiting slots from the VOC — this is to say that the instructions that would occupy the waiting slots during single-threaded execution are committed as well.

Second, we need to compute per-thread MLP under SMT execution. Memory-Level Parallelism (MLP) [6] is defined as the number of outstanding long-latency loads if at least one long-latency load is outstanding. This is measured using a single counter by monitoring the number of allocated MSHR entries.

Third, we need to estimate the amount of MLP under single-threaded execution. To this end, we introduce a per-thread back-end miss event table (BMT) which keeps track of the dependencies between long-latency loads in a very concise structure. The BMT has as many rows as the processor supports outstanding long-latency loads; a BMT row is allocated when a long-latency load is committed. Each BMT row holds the following:

(i) The *Long-latency Load ID (LLID)* records the instruction ID (modulo ROB_size) of the (committed) long-latency load.

(ii) The *Output Register Bit Vector (ORBV)* keeps track of all the architectural registers written by the load and its dependent instructions. The ORBV is initialized with the output register of the long-latency load, i.e., a '1' is written at the corresponding bit position, and all the other bits are reset. In case a committed instruction depends on a prior long-latency load — this is done by comparing the load's ORBV with the instruction's input registers — the load's ORBV is updated with a '1' at the bit position corresponding to the committed instruction's output register. In case the committed instruction does not depend on the prior long-latency load, a '0' is written at the bit position corresponding to the committed instruction's output register.

(iii) The *Dependent Instructions Counter (DIC)* counts the number of committed instructions that depend on the long-latency load — dependencies are determined through the ORBV.

(iv) The *Dependent Load Pointer (DLP)* to the long-latency load that the load depends upon, if applicable.

Dispatch would stall in single-threaded execution upon a long-latency load if the ROB fills up or if the issue queue fills up. These two conditions can be determined from the BMT: (i) the ROB would fill up if the delta between the

ID of the next to commit instruction and the LLID for the long-latency load at the BMT head equals the ROB size; (ii) the issue queue fills up if the number of dependent instructions (counted by the DIC) equals the issue queue size. Under one of these long-latency load stall conditions, we estimate single-threaded MLP by counting the number of long-latency loads that are independent of the leading long-latency load miss; this is done by counting the number of long-latency loads in the BMT that do not depend (either directly or indirectly) on the leading load miss. The ratio of the MLP under SMT execution with the MLP under single-threaded execution then yields a correction factor for the long-latency miss event cycle component, as described earlier.

3.5 Sharing misses

To account for the additional sharing misses in the branch predictor and caches/TLBs, we introduce a sharing miss correction mechanism, i.e., we estimate the number of sharing misses at run time due to multi-threading and then rescale the respective cycle components.

For set-associative caches, we estimate the number of sharing misses through set sampling. We sample a limited number of sets (we use the dynamic set sampling scheme by Qureshi et al. [20]) and maintain a per-thread tag directory that maintains the (per-thread) tags for these sets. When accessing a sampled set in the shared cache, we also access the per-thread tag directories. In case of a miss in the shared cache and a hit in the per-thread tag directory, the miss is considered a sharing miss. In our setup, we sample only 16 sets out of 4K sets for the L3 cache.

For fully-associative caches, such as a TLB, we maintain a fully-associative per-thread tag directory that maintains the tags of the past few missed cache lines. Similarly as above, in case of a miss in the shared cache and a hit in the per-thread tag directory, the miss is assumed to be a sharing miss. In our setup, the tag directory maintains 16 tags out of the 512 entries for the D-TLB.

We found sampling to be not as effective for shared branch predictors as for caches, and we therefore tag each branch predictor table entry with an ID of the thread which most recently updated the entry. In case of a branch misprediction and the tagged thread ID does not match the current thread ID, the branch misprediction is assumed to be a sharing miss. Tagging each branch predictor entry with thread IDs requires $\log_2\lceil N \rceil$ bits per entry with N the number of SMT threads.

3.6 Hardware cost

The above description of the per-thread cycle accounting architecture is functional rather than hardware implementation oriented. The reason is that the counter architecture can be implemented completely in hardware, but could also be partially implemented in software in which case the hardware would only need to count some of the raw events which are then processed by software. The decision of a pure hardware versus a hybrid hardware/software implementation depends on the counter architecture usage versus hardware cost. If the goal is to report per-thread alone execution times to system software, a hybrid hardware/software implementation is a viable solution which reduces the hardware cost; however, if a per-thread progress aware fetch policy is the goal, the counter architecture needs to be implemented in hardware. We now determine the cost for implementing the counter architecture completely in hardware; a hybrid hardware/software implementation will incur a smaller hardware cost.

The FMT requires $c \cdot n \cdot b$ bits with c the number of counters per row, n the number of bits per counter, and b the number of outstanding branches in the processor. This is $5 \cdot 10 \cdot 16 = 800$ bits in our implementation,

The hardware cost for the back-end miss event counter architecture is as follows: (i) the VOCs require $\lceil log_2 ROB \rceil$ bits per thread; (ii) the SMT MLP estimator requires one counter per thread; (iii) the BMT requires $n \cdot (2 \cdot log_2 ROB + log_2 n + 64)$ bits in total, with n the number of outstanding long-latency loads, $2 \cdot log_2 ROB$ bits for the LLID and DIC counters, $log_2 n$ bits for the DLP, and 64 bits for the ORBV (assuming 64 architectural registers); assuming 16 outstanding long-latency loads and a 256-entry ROB, the BMT incurs 1344 bits of storage.

The hardware cost for estimating the number of sharing misses in the branch predictor equals 2Kbits, and equals 4.5Kbits in total for the caches and TLBs.

The total hardware cost for the counter architecture thus equals 8.6Kbits or slightly more than 1Kbyte.

4. Experimental setup

We use the SPEC CPU2000 benchmarks in this paper with their reference inputs. These benchmarks are compiled for the Alpha ISA using the Compaq C compiler (`cc`) version V6.3-025 with the `-O4` optimization option. For all of these benchmarks we select 200M instruction (early) simulation points using the SimPoint tool [22]. We use 36 randomly selected two-thread and 30 four-program workloads, however, most of these workloads are memory-intensive in order to stress our cycle accounting architecture, i.e., we found the largest errors to appear for memory-intensive workloads.

We use the SMTSIM simulator [26] in all of our experiments, and we assume a 4-wide superscalar out-of-order SMT processor as shown in Table 1. We assume a shared reorder buffer, issue queue and rename register file unless mentioned otherwise; the functional units are always shared among the co-executing threads. We assume the ICOUNT fetch policy unless mentioned otherwise.

5. SMT Cycle Stacks: Evaluation

We now evaluate the accuracy of the per-thread SMT cycle stacks computed using the proposed counter architecture by

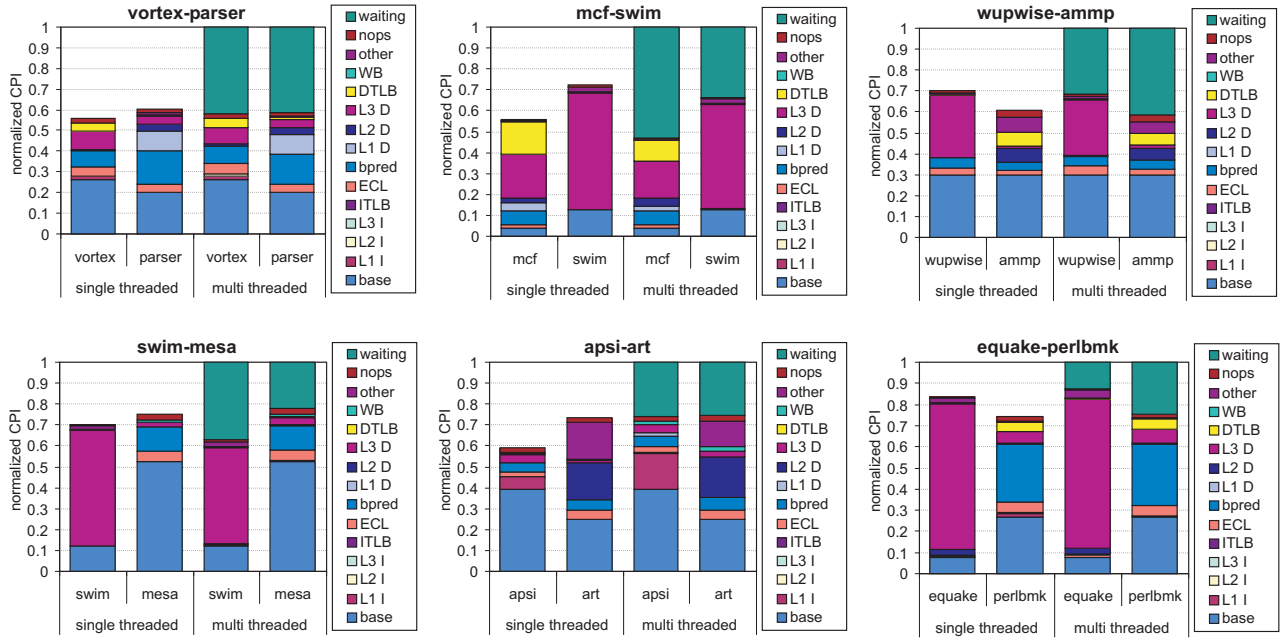

Figure 3. Evaluating SMT cycle stacks versus single-threaded cycle stacks for six example workloads assuming dynamic resource partitioning and the ICOUNT fetch policy.

parameter	value
fetch policy	ICOUNT 2.4
pipeline depth	14 stages
(shared) reorder buffer size	256 entries
instruction queues	96 entries in both IQ and FQ
rename registers	200 integer and 200 floating-point
processor width	4 instructions per cycle
functional units	4 int ALUs, 2 ld/st units and 2 FP units
branch misprediction penalty	11 cycles
branch predictor	2K-entry gshare
branch target buffer	256 entries, 4-way set associative
write buffer	24 entries
L1 instruction cache	64KB, 2-way, 64-byte lines
L1 data cache	64KB, 2-way, 64-byte lines
unified L2 cache	512KB, 8-way, 64-byte lines
unified L3 cache	4MB, 16-way, 64-byte lines
instruction/data TLB	128/512 entries, fully-assoc, 8KB pages
cache hierarchy latencies	L2 (11), L3 (35), MEM (350)

Table 1. The baseline SMT processor configuration.

comparing them against the cycle stacks computed under single-threaded execution, called the ST cycle stacks. This is done as follows. We run each multi-threaded workload for 400M instructions, compute the per-thread SMT cycle stacks as described in the previous sections, and record the number of instructions executed so far for each thread. We then run each thread in single-threaded mode for as many instructions as it ran under SMT execution, and compute the single-threaded cycle stacks following the method by Eyerman et al. [12].

Cycle component prediction. Figure 3 compares the SMT cycle stacks against the ST cycle stacks for six two-program workloads; these example workloads were chosen to represent an interesting sample of workloads. Each graph shows the two ST cycle stacks on the left and the two SMT cycle stacks on the right; the cycle stacks are normalized to the SMT cycle stacks. The cycle stack prediction errors stem from a number of sources. The branch misprediction cycle component is sometimes underestimated; the reason is that the critical path to the mispredicted branch tends to be smaller under SMT execution than under single-threaded execution because fewer critical path instructions are left unissued in the issue queue by the time the mispredicted branch is dispatched, i.e., the branch resolution time is shorter under SMT execution. Another source of error is that the counter architecture does not account for bandwidth sharing. For example, for the apsi-art workload (see middle bottom graph in Figure 3), the error in the L1 I-cache cycle component for apsi is due to limited bandwidth between the L1 and L2 caches under SMT execution. Estimating the L3 miss cycle component incurs two difficulties: (i) the single-threaded MLP estimate tends to be an overestimation because it does not account for MLP limiters such as branch mispredictions between two long-latency loads that depend on the first long-latency load, and long-latency I-cache misses intervening two long-latency loads; and (ii) the SMT MLP calculation may be inaccurate because it also counts MLP along mispredicted paths. Estimating the number of sharing misses complicates D-TLB cycle component computation. The largest cycle component errors that we observed across all workload mixes are as follows: 18% for the L1 I-cache component, 12% for the L2 D-cache component, 13% for the L3 D-cache component, 24% for the D-TLB component and 12% for the 'other' component.

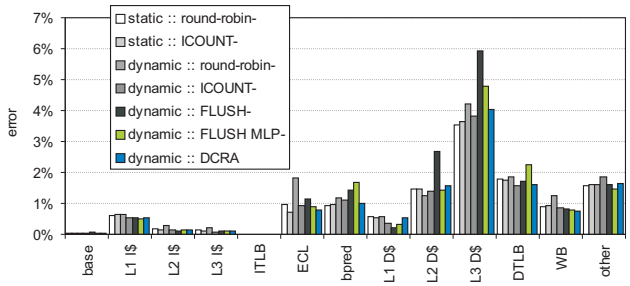

Figure 4. The average absolute cycle component prediction error across all workloads and six SMT processor fetch policies.

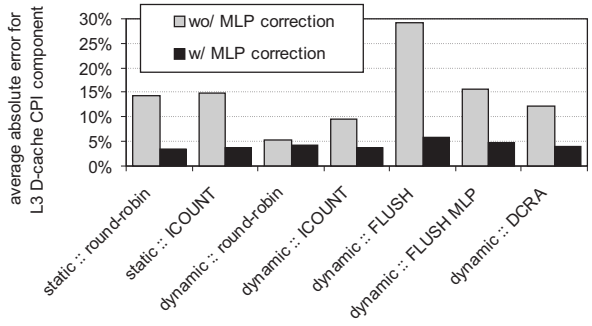

Figure 5. The average absolute L3 D-cache cycle component prediction error with and without MLP correction.

Figure 4 shows the average cycle component prediction error for seven resource sharing and fetch policies: (i) static partitioning and the round-robin fetch policy, (ii) static partitioning and the ICOUNT fetch policy [28], (iii) dynamic partitioning and round-robin, (iv) dynamic partitioning and ICOUNT, (v) dynamic partitioning and flush [27] (flush threads that experience a long-latency load), (vi) dynamic partitioning and MLP-aware flush [10] (flush long-latency load threads while preserving MLP), and (vii) dynamic partitioning through resource allocation, or DCRA [3]. Static resource partitioning assumes a private reorder buffer, issue queues and rename register file, but the functional units are shared; dynamic resource partitioning assumes that all of these resources are shared. Figure 4 shows that the cycle accounting architecture is accurate across multiple SMT fetch policies with average cycle component prediction errors of a few percent. The largest average error (no more than 6%) is observed for the long-latency load (L3) cycle component.

Importance of MLP correction. Figure 5 illustrates the importance of MLP correction for estimating the long-latency load (L3) cycle component: the average absolute prediction error is shown for the L3 D-cache cycle component both with and without MLP correction — the other cycle components are not affected by MLP correction. The average error drops below 6% across all fetch policies with MLP correction. Without MLP correction, the average error

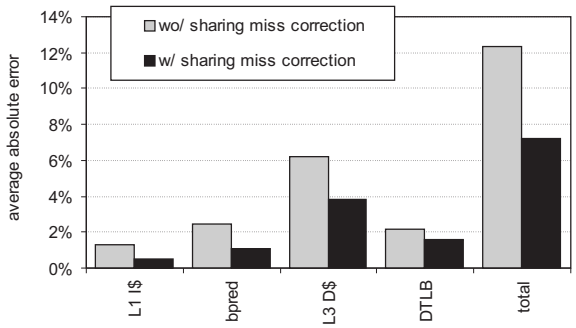

Figure 6. Cycle component prediction errors with and without sharing miss correction.

is typically higher and can be as high as 29% for the flush policy. The reason for the high error without MLP correction is the discrepancy in exploitable MLP under single-threaded execution versus SMT execution: a thread in single-threaded execution gets the entire ROB for exposing MLP, whereas a thread in an SMT processor typically does not. This is magnified in the flush policy which flushes a thread that experiences a long-latency load, i.e., the flushed thread does not get to expose MLP, which explains the high error without MLP correction. The MLP correction mechanism corrects the observed MLP under SMT execution with the estimated MLP under single-threaded execution.

Importance of sharing miss correction. Figure 6 quantifies the importance of sharing miss correction. The cycle component prediction error is reduced substantially through sharing miss correction, e.g., the branch misprediction cycle component reduces from 2.5% to 1.1%, and the L3 cache cycle component reduces from 6.2% to 3.8%. Collectively, sharing miss correction improves alone execution time estimation from 12.3% down to 7.2%.

Per-thread progress prediction. So far, we were concerned with estimating cycle components. Adding the base cycle component to the miss event cycle components yields the overall alone cycle count, and dividing it by the number of instructions executed yields the overall alone CPI. In Figure 7, the CPI prediction by the cycle accounting architecture is compared against the CPI obtained from single-threaded execution. These graphs show that the estimated execution time correlates well with the measured single-threaded execution time; the average absolute prediction error equals 7.2%; we observe similar results for the other resource sharing policies: static partitioning and round-robin (7.2%), static partitioning and ICOUNT (7.1%), dynamic partitioning and round-robin (8.4%), flush (8.5%), MLP-aware flush (8.5%), and DCRA (6.9%). Figure 8 shows similar results for four-program workloads; the average alone execution time prediction error equals 11.7%.

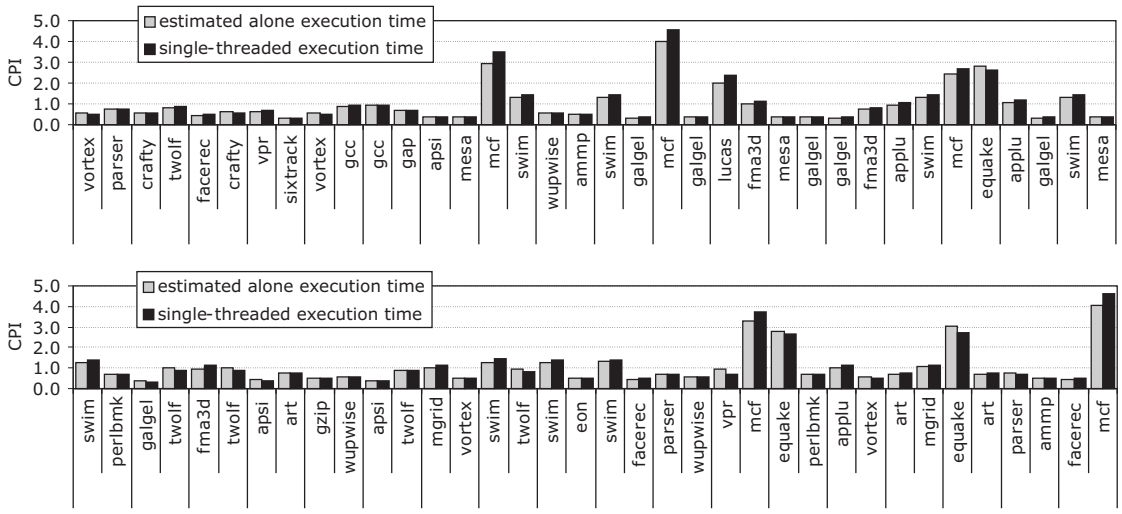

Figure 7. Estimated alone execution time versus measured single-threaded execution time for the two-program workloads; each two-program workload is separated by a vertical line.

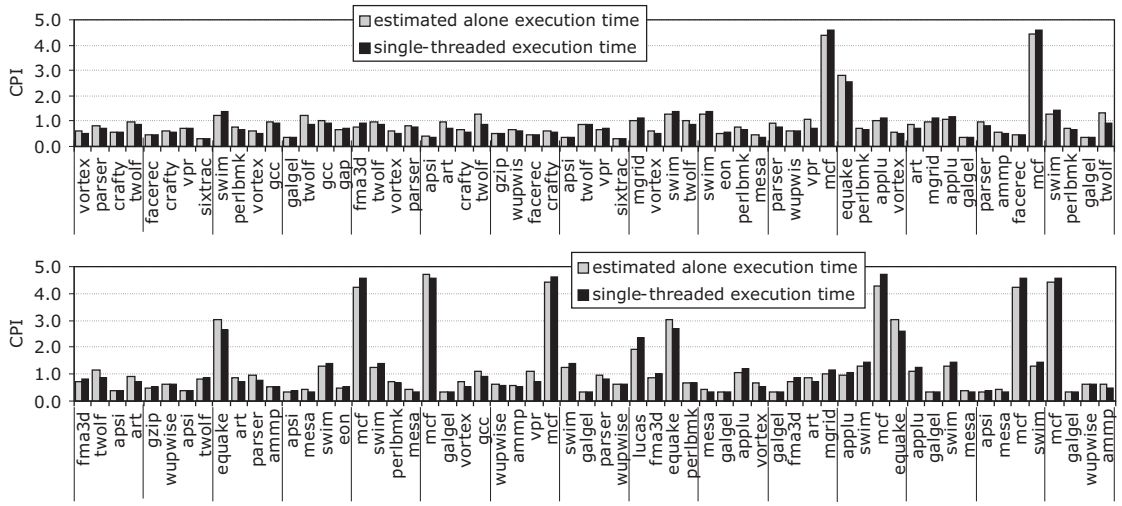

Figure 8. Estimated alone execution time versus measured single-threaded execution time for the four-program workloads; each four-program workload is separated by a vertical line.

6. Applications

There are at least two important applications for the proposed cycle accounting architecture: alone execution time reporting and thread-progress aware fetch policies. We now discuss both in more detail.

6.1 Alone execution time reporting

The first application of SMT cycle accounting is to report per-thread alone execution times to system software. This means that upon the termination of a timeslice, the counter architecture reports the alone execution times for each of the co-running threads. The alone execution time provides a more accurate picture of per-thread progress than the actual execution time (i.e., the timeslice) as currently assumed by system software schedulers. Schedulers leveraging the alone execution time instead of the actual execution time may be more effective at providing quality-of-service (QoS), priorities and performance isolation.

In addition, the alone execution time enables accurate per-CPU-hour billing in SMT computing environments. Existing SMT processors have no means of tracking or controlling per-thread progress indicators. Billing a program that gets seriously slowed down by its co-running program(s) for its actual execution time is unfair. The counter architecture may alleviate this concern: each program gets a bill proportional to its alone execution time.

6.2 Thread-progress aware fetch policies

SMT cycle accounting also enables a new class of *thread-progress aware* fetch policies. Thread-progress aware fetch policies continuously monitor per-thread progress in hard-

ware and drive the fetch policy based on the per-thread progress indicators in order to achieve a pre-set per-thread performance target. In other words, a thread-progress aware fetch policy controls the progress of the threads co-executing on an SMT processor while utilizing left-over instruction bandwidth to optimize system throughput. The key feature thread-progress aware fetch policies offer is that they enable imposing system software priorities at the hardware level; and this may increase the responsiveness of the system.

To this end, we introduce *Thread Progress Registers (TPRs)*, set by system software, that determine the fraction of the total execution cycles that should be devoted to each thread. The sum of all TPRs should not be larger than one, but may be lower. For example in a two-thread system, a TPR configuration of 75%–25% means that thread 1 and thread 2 should get at least 75% and 25% of the total execution cycles, respectively; the additional instruction bandwidth due to overlap effects during SMT execution can be distributed among both threads for improving system throughput. A 50%–50% TPR configuration corresponds to a fair fetch policy in which both threads should make equal progress. The meaning of the TPRs is consistent with the general understanding of timeslices in system software for managing thread priorities, QoS and performance isolation. For example, a 75%–25% TPR configuration could be realized in a single-threaded processor system through time multiplexing by giving 75% of the timeslices to thread 1 and 25% of the timeslices to thread 2. An SMT processor with a thread-progress aware fetch policy and TPRs can provide this same abstraction to system software while achieving significantly better system throughput.

A thread-progress aware fetch policy with TPRs operates as follows. For each thread i, the hardware thread priority controller computes the relative progress indicator P_i:

$$P_i = (C_{base,i} + C_{miss\ event,i}) - C_{total} \cdot TPR_i,$$

with C_{total} the total cycle count, TPR_i the thread's TPR, and $C_{base,i} + C_{miss\ event,i}$ the thread's base plus miss event cycle count (the thread's alone execution time). The fetch policy then gives priority to the thread with the lowest P_i at run time, i.e., the thread that is lagging behind its target performance the most. In order to improve system throughput, we include the following two optimizations: (i) when the V-ROB gets exhausted, the thread is stalled — this is to prevent the thread from clogging resources on a long-latency load — and (ii) when dispatch stalls on a resource stall (due to reorder buffer, issue queue or rename register file exhaustion), the thread with the largest V-ROB occupancy is flushed for half the difference in V-ROB instructions with the smallest V-ROB thread — this is to de-allocate part of the clogged resources.

The evaluation of the proposed thread-progress aware fetch policy is done in two steps. We first assume that both threads are given the same priority, i.e., $TPR_1 = TPR_2 = 50\%$. This enables comparing the thread-progress aware fetch policy against existing fetch policies. We subsequently consider a scenario in which one thread gets a higher priority than the other, i.e., $TPR_1 = 75\%$ and $TPR_2 = 25\%$. This will enable evaluating whether thread-progress aware fetch policy can control per-thread progress.

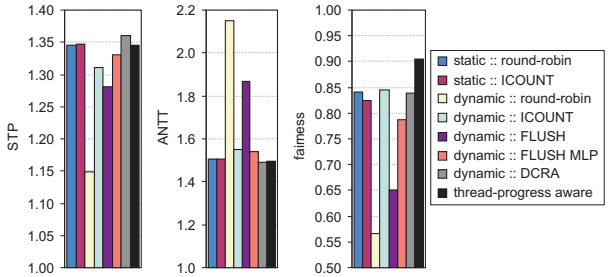

Figure 9. Comparing the thread-progress aware fetch policy against the other policies in terms of STP, ANTT and fairness.

Figure 9 shows system throughput (STP)[1] [11], average normalized turnaround time (ANTT)[2] [11], and fairness[3] [11, 15] for the thread-progress aware fetch policy against the other fetch policies. Whereas the thread-progress aware fetch policy achieves a comparable STP and ANTT, it achieves a substantially better fairness — more than 7% increase in fairness compared to round robin (under static resource partitioning), ICOUNT and DCRA which are the fairest policies among all the other policies.

Figure 10 evaluates the thread-progress aware policy for the $TPR_1 = 75\%$ and $TPR_2 = 25\%$ scenario. For all of the threads, the fetch policy achieves per-thread performance above its TPR: both threads achieve a performance above the 75% and 25% thresholds, respectively. The left-over computation bandwidth above the TPR thresholds is distributed across both threads; the excess computation bandwidth improves system throughput on SMT processors not realizable through time multiplexing tasks on single-threaded processors.

7. Related Work

Counter architectures. Modern microprocessors offer hardware performance monitors for counting dynamic processor events such as the number of cycles, instructions, cache misses, etc. and/or CPI stacks [19]. Eyerman et al. [12] propose a top-down approach to architecting CPI stack counter architectures. Some proposals such as ProfileMe [8] and shotgun profiling [14] use a combination of hardware and software for computing CPI stacks and per-instruction exe-

[1] STP is a higher-is-better metric and equals the weighted speedup metric by Snavely and Tullsen [23].

[2] ANTT is a lower-is-better metric and is the inverse of the hmean metric by Luo et al. [18].

[3] Fairness is a higher-is-better metric.

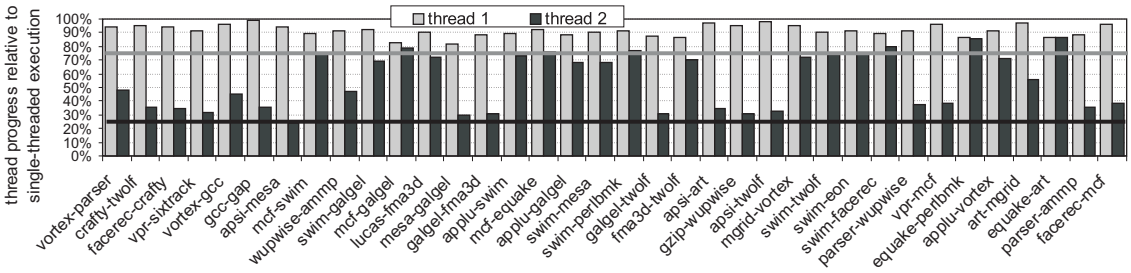

Figure 10. Evaluating SMT performance guarantees under a $TPR_1 = 75\%$ and $TPR_2 = 25\%$ scenario.

cution profiles. None of these counter architectures aim at SMT processors though.

Resource partitioning in SMT processors. The partitioning of resources such as the reorder buffer, issue queues, rename registers, etc. can be done either statically [21] or dynamically. Dynamic resource partitioning provides more flexibility but requires a so called fetch policy, such as ICOUNT [28], flush [27], MLP-aware flush [10], or DCRA [3]. The ICOUNT fetch policy, which is the basic policy on which the others are built, strives at having an equal number of instructions from all threads in the front-end pipeline and issue queues — motivated by Little's law as described by Emer [9]. In other words, a thread that makes fast progress under single-threaded mode will make faster progress under ICOUNT than a thread that makes slow single-threaded progress. One could thus argue that ICOUNT is thread-progress aware. However, this assumes that co-executing threads do not affect each other's performance, which we find not to be true due to I-cache misses (e.g., a thread will make faster single-thread progress while an I-cache miss from another co-executing thread is being resolved), D-cache misses (e.g., loss in per-thread MLP during SMT execution), and resource sharing in the branch predictor and caches. The proposed counter architecture on the other hand takes these inter-thread performance interactions into account while tracking per-thread progress. In addition, none of the prior fetch policies enable enforcing system software priorities in hardware.

QoS management in SMT processors. A number of studies have been done on improving QoS in SMT processors.

Cazorla et al. [2, 4] target QoS in SMT processors through resource allocation. They propose a system that samples single-threaded IPC, and dynamically adjusts the resources to achieve a pre-set percentage of the single-threaded IPC. The single-threaded IPC sampling overhead incurs a 5% runtime overhead; the SMT cycle accounting architecture proposed here eliminates this sampling overhead by computing single-threaded progress online with no runtime overhead.

Cota-Robles [7] describes an SMT processor architecture that combines OS priorities with thread efficiency heuristics (outstanding instruction counts, number of outstanding branches, number of data cache misses) to provide a dynamic priority for each thread scheduled on the SMT processor. SMT cycle accounting enables a per-thread progress indicator as input to the fetch policy rather than a thread efficiency heuristic.

The IBM POWER5 [1] implements a software-controlled priority scheme that controls the per-thread dispatch rate. Software-controlled priorities are independent of the operating system's concept of thread priority and are used for temporarily increasing the priority of a process holding a critical spinlock, or for temporarily decreasing the priority of a process spinning for a lock, etc.

Snavely et al. [24] study symbiotic job scheduling on SMT processors that strive at maximizing system throughput while satisfying job priorities. Jain et al. [16] study symbiotic job scheduling of soft real-time applications on SMT processors. Fedorova et al. [13] find that non-work-conserving scheduling, i.e., running fewer threads than the SMT processor allows, can improve system performance; they use an analytical model to find cases where a non-work-conserving policy is beneficial. Making SMT scheduling aware of single-threaded performance may yield even more effective co-execution schedules that better balance throughput and job turnaround time.

SOE processors. Gabor et al. [15] propose fairness enforcement based on a cycle accounting mechanism for coarse-grain switch-on-event (SOE) multithreaded processors. The cycle accounting architecture estimates single-thread performance for each of the threads had each of them been executed alone, while they are running in SOE multithreading. Our work concerns with cycle accounting for SMT processors rather than SOE processors which poses a number of additional challenges because of the tighter interactions between co-executing threads on an SMT processor.

8. Conclusion

This paper proposed an SMT cycle accounting architecture for estimating per-thread alone execution time during SMT execution. The proposed cycle accounting architecture can be employed with multiple fetch policies, and estimates per-thread alone execution time within 7.2% on average for two-program workloads and 11.7% for four-program workloads; the counter architecture incurs a reasonable hardware cost of around 1KB of storage.

The cycle accounting architectures enables a number of applications. For one, a thread's alone execution time is more accurate than the actual execution time (timeslice) assumed by system software, which may make a time multiplexing based scheduler more effective. Second, it enables a new class of thread-progress aware fetch policies controlled by system software that guarantee a performance level to a thread irrespective of the other threads that happen to be running simultaneously; left-over instruction bandwidth is used to optimize system throughput. In addition, a thread-progress aware fetch policy enables imposing system software priorities at the hardware level.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. Stijn Eyerman and Lieven Eeckhout are Postdoctoral Fellows with the Fund for Scientific Research in Flanders (Belgium) (FWO-Vlaanderen).

References

[1] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero. Software-controlled priority characterization of POWER5 processor. In *ISCA*, pages 415–426, June 2008.

[2] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, July 2006.

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

[4] F. J. Cazorla, A. Ramirez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, 24(4):24–31, July 2004.

[5] S. Choi and D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In *ISCA*, pages 239–250, June 2006.

[6] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA*, pages 76–87, June 2004.

[7] E. Cota-Robles. *Priority Based Simultaneous Multi-Threading*, Dec. 2003. United States Patent No. 6,658,447 B2.

[8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. *ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors. In *MICRO*, Dec. 1997.

[9] J. Emer. EV8: The post-ultimate alpha. Keynote presentation at PACT, Sept. 2001.

[10] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *HPCA*, pages 240–249, Feb. 2007.

[11] S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[12] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, pages 175–184, Oct. 2006.

[13] A. Fedorova, M. Seltzer, and M. D. Smith. A non-work-conserving operating system scheduler for SMT processors. In *WIOSCA, in conjunction with ISCA*, June 2006.

[14] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization*, 1(3):272–304, Sept. 2004.

[15] R. Gabor, S. Weiss, and A. Mendelson. Fairness enforcement in switch on event multithreading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3):34, Sept. 2007.

[16] R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium*, pages 134–145, Dec. 2002.

[17] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, pages 338–349, June 2004.

[18] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, Nov. 2001.

[19] A. Mericas. Performance monitoring on the POWER5 microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2006.

[20] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, pages 167–177, June 2006.

[21] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *PACT*, pages 15–26, Sept. 2003.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57, Oct. 2002.

[23] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *ASPLOS*, pages 234–244, Nov. 2000.

[24] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS*, pages 66–76, June 2002.

[25] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, fifth edition, 2005.

[26] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, Dec. 1996.

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

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

[29] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, June 1995.