Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance

Gautham K.Dorai and Donald Yeung
Department of Electrical and Computer Engineering and
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

Simultaneous Multithreading (SMT) processors achieve high processor throughput at the expense of single-thread performance. This paper investigates resource allocation policies for SMT processors that preserve, as much as possible, the single-thread performance of designated "foreground" threads, while still permitting other "background" threads to share resources. Since background threads on such an SMT machine have a near-zero performance impact on foreground threads, we refer to the background threads as *transparent threads*. Transparent threads are ideal for performing low-priority or non-critical computations. Application domains that map naturally onto transparent threads include scheduling of low-priority processes in a multiprogrammed workload, sub-ordinate multithreading, and on-line performance monitoring.

To realize transparent threads, we propose three mechanisms for maintaining the transparency of background threads: slot prioritization, background thread instruction-window partitioning, and background thread flushing. In addition, we propose three mechanisms to boost background thread performance without sacrificing transparency: aggressive fetch partitioning, foreground thread instruction-window partitioning, and foreground thread flushing. We implement our mechanisms on a detailed simulator of an SMT processor, and evaluate them using 7 benchmarks, 6 of which belong to the SPEC CPU2000 suite. Our results show when cache and branch predictor interference are factored out, background threads introduce less than 1% performance degradation on the foreground thread. Furthermore, maintaining the transparency of background threads reduces their throughput by only 22% relative to an equal priority scheme. In addition to evaluating our transparent thread mechanisms, we also study *Transparent Software Prefetching* (TSP), an implementation of software data prefetching using transparent threads. TSP achieves a 9.17% gain across our SPEC benchmarks, whereas conventional software prefetching degrades performance by 0.8%. The performance advantage of TSP comes from its 1.16% overhead, compared to a 17.87% overhead incurred by conventional software prefetching.

1 Introduction

Simultaneous multithreading (SMT) processors achieve high processor throughput by exploiting ILP between independent threads as well as within a single thread. The increased processor throughput provided by SMT, however, comes at the expense of single-thread performance. Because multiple threads share hardware resources simultaneously, individual threads get fewer resources

This research was supported in part by NSF Computer Systems Architecture grant CCR-0093110 and NSF CAREER Award CCR-0000988.

than they would have otherwise received had they been running alone. Furthermore, existing resource allocation policies, such as ICOUNT [24] and FPG [12], favor threads with high ILP, steering resources to threads whose instructions pass through the pipeline the most efficiently. Threads that utilize processor resources less efficiently receive fewer resources and run even slower.

Our work investigates resource allocation policies for SMT processors that preserve, as much as possible, the single-thread performance of designated high-priority or "foreground" threads, while still permitting other low-priority or "background" threads to share resources. Our approach allocates resources to foreground threads whenever they can use them and regardless of how inefficiently those resources might be used, thus permitting foreground threads to run as fast as they would have run standalone. At the same time, we only allocate *spare* resources to background threads that foreground threads would have otherwise left idle, thus allowing background threads to use processor resources without degrading foreground thread performance. Since the background threads on such an SMT machine are imperceptible to the foreground threads (at least from a resource sharing standpoint), we refer to the background threads as *transparent threads*.

Transparent threads are ideal for performing low-priority or non-critical computations. Several applications of multithreading involve such low-priority computations, and thus map naturally onto transparent threads:

Process Scheduling. Transparent threads can assist in scheduling multi-programmed workloads onto SMT processors. When a latency-sensitive process enters a multi-programmed workload (for example, when an interactive process receives an event), all non-latency-sensitive processes can be down-graded to run as transparent threads. During the time that the latency-sensitive or foreground process is active, the transparent threads yield all processor resources necessary for the foreground process to run as fast as possible. At the same time, the transparent threads are not shut out completely, receiving any resources that the foreground process is unable to use.

Sub-ordinate Multithreading. Transparent threads can support sub-ordinate multithreading [5]. Sub-ordinate threads perform computations on behalf of a primary thread to increase its performance. Recently, there have been several proposals for sub-ordinate multithreading. Sub-ordinate threads have been used to perform prefetching [11, 7, 25, 17] (also known as *pre-execution*), cache management [8], and branch prediction [5]. Unfortunately, the benefit of these techniques must be weighed against their cost. If the overhead of sub-ordinate computation outweighs the optimization benefit, then applying the optimization may reduce rather than increase performance. For this reason, detailed profiling is necessary to determine when optimizations are profitable so that optimizations can be applied selectively to minimize overhead.

Transparent threads enable sub-ordinate multithreading optimizations to be applied *all the time*. Since transparent threads never take resources away from foreground threads, sub-ordinate threads that run as transparent threads incur zero overhead; hence, optimizations are always profitable, at worst providing zero gain. With transparent threading support, a programmer (or compiler) could apply sub-ordinate threading optimizations blindly. Not only does this relieve the programmer from having to perform profiling, but it also increases the optimization coverage resulting in higher performance.

Performance Monitoring. Finally, transparent threads can execute profiling code. Profiling systems often instrument profile code directly into the application [3, 10, 20]. Unfortunately, this can result in significant slow-down for the host application. To minimize the impact of the instrumentation code, it may be possible to perform the profiling functionality inside transparent threads. Similar to sub-ordinate multithreading, profile-based transparent threads would not impact foreground thread performance, and for this reason, could enable the use of profiling code all the time.

This paper investigates the mechanisms necessary to realize transparent threads for SMT processors. We identify the hardware resources inside a processor that are critical for single-thread performance, and propose techniques to enable background threads to share them transparently with foreground threads. In this paper, we study the transparent sharing of two resources that impact performance the most: instruction slots and instruction buffers. We also discuss transparently sharing a third resource, memories, but we do not evaluate these solutions in this paper. Next, we propose techniques to boost transparent thread performance. Using our basic resource sharing mechanisms, transparent threads receive hardware resources only when they are idle. Under these conservative assumptions, transparent threads can exhibit poor performance. We propose additional techniques that detect when resources held by foreground threads are not critical to their performance, and aggressively reallocate them to transparent threads.

To evaluate the effectiveness of our techniques, we undertake an experimental evaluation of transparent threads on a detailed SMT simulator. Our evaluation proceeds in two steps. First, we study transparent threads in the context of multi-programming. Our multi-programming experiments illustrate the interactions of diverse workloads and identify the most important mechanisms for enforcing transparency under these diverse interactions. Second, we study *Transparent Software Prefetching* (TSP), an implementation of software data prefetching of affine array references [14] using transparent threads. By off-loading prefetch code onto transparent threads, we achieve virtually zero-overhead software prefetching. We show this enables software prefetching for all candidate memory references, thus eliminating the need to perform profiling a priori to identify cache-missing memory references, as is required in previous software prefetching studies.

The rest of the paper is organized as follows. Section 2 presents our mechanisms in detail. Next, Section 3 describes our simulation framework used to perform the experimental evaluation. Then, Section 4 studies transparent threads in the context of multi-programming and Section 5 studies TSP. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2 Transparent Threads

This section presents the mechanisms necessary to support transparent threads. First, Section 2.1 discusses the impact of resource sharing on single-thread performance in an SMT processor. Next, Section 2.2 presents the mechanisms for transparently sharing two classes of resources, instruction slots and buffers, and discusses possible solutions for transparently sharing a third class, memories. Finally, Section 2.3 presents the mechanisms for boosting transparent thread performance.

2.1 Resource Sharing

Figure 1 illustrates the hardware resources in an SMT processor pipeline. The pipeline consists of three major components: fetch hardware (multiple program counters, a fetch unit, a branch predictor, and an I-cache), issue hardware (instruction decode, register rename, instruction issue queues, and issue logic), and execute hardware (register files, functional units, a D-cache, and a reorder buffer). Among these hardware resources, three are dedicated. Each thread has its own program counter and return stack (the return stack is part of the branch predictor module in Figure 1). In addition, each thread effectively has its own set of register files as well since the integer and floating point register files, while centralized, are large enough to hold the architected registers from all contexts simultaneously. All other hardware resources are shared between threads.

Simultaneously executing threads increase processor throughput by keeping shared hardware

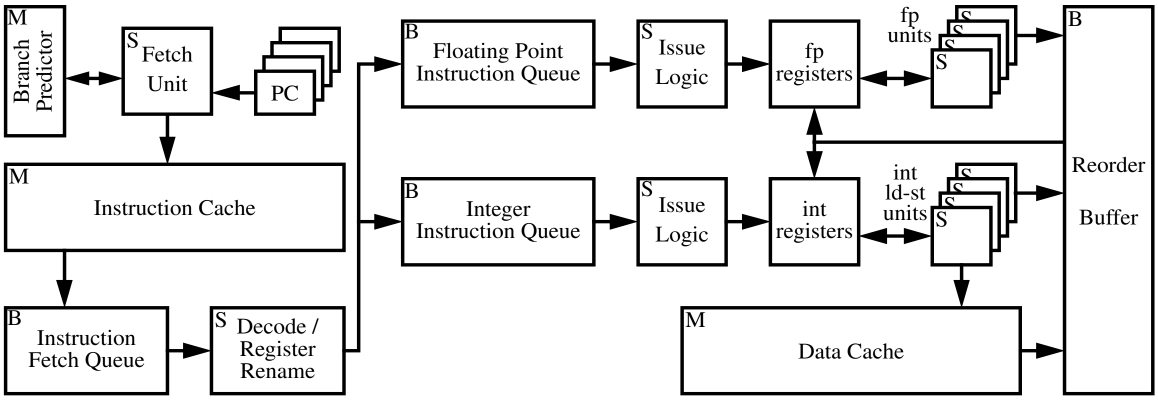

Figure 1: SMT processor hardware (diagram adapted from [24]). Each shared hardware resource is labeled with a letter signifying one of three resource classes: instruction slots (S), instruction buffers (B), and memories (M).

resources utilized as often as possible, but degrade each others' performance by competing for these resources. The goal of transparent threading, therefore, is to allocate a shared resource to a background thread only when it is not competing with a foreground thread for the same resource. To provide more insight, we group the shared hardware resources into three classes–instruction slots, instruction buffers, and memories–and discuss their resource allocation properties. Each shared resource in Figure 1 is labeled with its resource class, using the labels "S," "B," and "M," respectively.

Instruction slots are pipeline stages. The fetch, decode, rename, issue, writeback, and commit stages contain instruction slots, typically equal in number to the width of the machine. In addition, functional units also contain slots, typically one per functional unit per cycle of latency (assuming a fully pipelined unit). Instruction buffers hold stalled instructions. Figure 1 shows four buffers: the instruction fetch queue holds fetched instructions waiting to be decoded and renamed, the integer and floating point instruction queues hold instructions waiting on operands and/or functional units, and the reorder buffer holds instructions waiting to commit. Finally, memories are cache structures. The I- and D-caches as well as the branch predictor tables in Figure 1 make up this category.

The ramifications for allocating a shared resource to a background thread depend on its resource class. Allocating an instruction slot to a background thread impacts the foreground thread on the *current cycle only*. Instructions normally occupy slots for a single cycle. While there are exceptions to this rule (for example a load instruction that suffers a cache miss in its data-cache access stage), we find these cases do not create resource conflicts frequently. Therefore, background threads can use instruction slots transparently as long as there is no conflict with the foreground thread on the cycle of allocation. In contrast, allocating an instruction buffer entry to a background thread potentially impacts the foreground thread on *future cycles*. Instructions typically occupy buffers for many cycles, particularly in the reorder buffer where instructions remain until all preceding instructions (including those performing long-latency operations) commit. Therefore, allocating a buffer entry to a background thread can cause resource conflicts with the foreground thread in the future even if the resource is idle on the cycle of allocation.

Compared to instruction slots and buffers, memory resource sharing has a less direct impact on foreground thread performance. Rather than taking an execution resource away from the foreground thread, the use of memory resources by a background thread potentially increases the number of performance-degrading events experienced by the foreground thread (*i.e.*, branch mispredictions and cache misses). Similar to instruction buffers, the impact does not occur at the time of use, but rather, at a point in the future.

2.2 Transparent Sharing Mechanisms

Having discussed the resource sharing problem in Section 2.1, we now present several mechanisms that permit background threads to share resources transparently with the foreground thread. We present one mechanism for sharing instruction slots, two for sharing instruction buffers, and finally, we discuss possible solutions for sharing memories.

Instruction Slots: Slot Prioritization. Since instruction slots are normally held for a single cycle only, we allocate an instruction slot to a background thread as long as the foreground thread does not require the slot on the same cycle. If the foreground thread competes for the same instruction slot resource, we give priority to the foreground thread and retry the allocation for the background thread on the following cycle.

As described in Section 2.1, every pipeline stage has instruction slots; however, we implement *slot prioritization* in the fetch and issue stages only. We find that prioritizing slots in additional pipeline stages does not increase the transparency of background threads. To implement slot prioritization in the fetch stage, we modify the SMT processor's fetch priority scheme. Our default scheme is ICOUNT [24]. When choosing the threads to fetch from on each cycle, we artificially increase the instruction count for all background threads by the total number of instruction window entries, thus giving fetch priority to foreground threads always regardless of the instruction count values. Background threads receive fetch slots only when the foreground thread cannot fetch, for example due to a previous I-cache miss or when recovering from a branch misprediction. Slot prioritization in the issue stage is implemented in a similar fashion. We always issue foreground thread instructions first; background thread instructions are considered for issue only when issue slots remain after all ready foreground thread instructions have been issued.

Instruction Buffers: Background Thread Instruction-Window Partitioning. Compared to instruction slots, transparently allocating instruction buffer resources is more challenging because resource allocation decisions impact the foreground thread on future cycles. It is impossible to guarantee at allocation time that allocating an instruction buffer entry to a background thread will not cause a resource conflict with the foreground thread. Determining this would require knowing for how long the background thread will occupy the entry as well as knowing the number of buffer entries the foreground thread will request in the future.

We propose two solutions for transparently allocating instruction buffers. First, we limit the maximum ICOUNT value for background threads. When a background thread reaches this instruction count limit, it is not allowed to consume fetch slots even if the foreground thread leaves some fetch slots idle. The background thread remains locked out of the fetch stage until its ICOUNT value drops. We call this scheme *background thread instruction-window partitioning.*

The background thread instruction-window partitioning scheme ensures that the total number of background thread instructions in the instruction fetch queue and the reorder buffer never exceeds its instruction count limit. Notice this does not guarantee that background threads never take instruction buffer resources away from the foreground thread. If the foreground thread tries to consume most or all of the instruction buffer resources, it can still "collide" with the background threads in the buffers and be denied buffer resources. However, this scheme limits the damage that background threads can inflict on the foreground thread. By limiting the maximum number of buffer entries allocated to background threads, a large number of entries can be reserved for the foreground thread.

Instruction Buffers: Background Thread Flushing. In our second scheme for transparently allocating instruction buffers, we permit background threads to occupy as many instruction buffer entries as they can (under the constraint that the foreground thread gets all the fetch slots it requests), but we pre-emptively reclaim buffer entries occupied by background threads when necessary. We call this mechanism *background thread flushing.*

Background thread flushing works in the following manner. First, we trigger background thread flushing whenever the foreground thread tries to allocate an instruction buffer entry but all entries of that type are filled. There are four instruction buffers, as shown in Figure 1, whose allocation can trigger flushing: the instruction fetch queue, the integer and floating point instruction queues, and the reorder buffer. Among these four instruction buffers, we have observed that reorder buffer contention is responsible for the most performance degradation in the foreground thread (in fact, contention for the other instruction buffers usually occurs when the reorder buffer is full). For simplicity, we trigger flushing only when the foreground thread is unable to allocate a reorder buffer entry.

Once flushing is triggered, we select a background thread to flush. We compare the ICOUNT values of all background threads and pick the thread with the largest value. From this thread, we flush the N youngest instructions in the reorder buffer, where N is the width of the machine. (If the background thread occupies fewer than N reorder buffer entries, we flush all of its entries). Any instructions in the integer or floating point instruction queues corresponding to flushed reorder buffer entries are also flushed. In addition, we flush all instruction fetch queue entries belonging to this thread. Finally, we roll back the thread's program counter and register file map to the youngest unflushed instruction. Notice our flushing mechanism is similar to branch misprediction recovery; therefore, most of the hardware necessary to implement it already exists. However, our mechanism requires checkpointing the register file map more frequently since we flush to an arbitrary point in the reorder buffer rather than to the last mispredicted branch. In Section 3, we will discuss techniques for reducing the cost of implementing background thread flushing.

Compared to background thread instruction-window partitioning, background thread flushing requires more hardware support; however, it potentially permits background threads to share instruction buffer resources more transparently. Background thread flushing guarantees the foreground thread always gets instruction buffer resources, using pre-emption to reclaim resources from background threads if necessary. At the same time, background thread flushing can provide higher throughput compared to background thread instruction-window partitioning. If the foreground thread does not use a significant number of instruction buffer entries, the background threads can freely allocate them because there is no limit on the maximum number of entries that background threads can hold.

Memories: Possible Solutions. As our results in Section 4 will show, sharing instruction slot and instruction buffer resources has the greatest impact on foreground thread performance, while sharing memories has a less significant performance impact. For this reason, we focus on the first two classes of resources, and we do not evaluate mechanisms for transparently sharing memories in this paper.

We believe memory resources, *e.g.* branch predictor tables and caches, can be transparently shared using similar approaches described above. One possible approach is to limit the maximum number of locations that a background thread can allocate in the memories. Memory resources are used by mapping an address to a memory location. For branch prediction, a combination of the branch address and a branch history pattern is typically used to index into the branch predictor table. For caches, a portion of the effective memory address is used to index into the cache. Consequently, utilization of the memory resources can be limited by modifying the mapping function and using a reduced number of address bits to form the index. Background threads can use the modified mapping function, hence using a fewer number of memory locations. Foreground threads can use the normal mapping function to access the full resources provided by the memories.

2.3 Transparent Thread Performance Mechanisms

In Section 2.2, we focused on maintaining background thread transparency; however, achieving high background thread performance is equally important. Unfortunately, as Section 4 will show, the resource sharing mechanisms presented in Section 2.2 can starve background threads, leading to poor performance. This section presents several additional mechanisms for increasing resource allocation to background threads without sacrificing transparency. We present one mechanism for increasing fetch slot allocation, and two mechanisms for increasing instruction buffer allocation.

Fetch Instruction Slots: Fetch Partitioning. The most important instruction slot resources are the fetch slots because the frequency with which a thread receives fetch slots determines its maximum throughput. As described in Section 2.2, fetch slot prioritization always gives priority to the foreground thread by artificially increasing the ICOUNT values of the background threads. Even though the foreground thread always gets priority for fetch, the background thread can still get a significant number of fetch slots if the SMT employs an aggressive *fetch partitioning scheme*.

The most basic fetch partitioning scheme is to permit only one thread to fetch every cycle, and to give all the fetch slots to that single thread. Assuming an ICOUNT fetch priority scheme, this basic fetch partitioning scheme is called ICOUNT.1.N [24], where N is the fetch width. Under ICOUNT.1.N with slot prioritization, background threads receive fetch slots only when the foreground thread cannot fetch at all. If the foreground thread fetches even a single instruction, all N fetch slots on that cycle are allocated to the foreground thread since only one thread can fetch per cycle. In our SMT processor model, we assume the only times the foreground thread cannot fetch are 1) if it has suffered an I-cache miss, in which case it stalls until the cache miss is serviced, or 2) if it has suffered a branch mispredict, in which case it stalls until mispredict recovery completes.

If instead of allowing only a single thread to fetch per cycle, multiple threads are allowed to fetch per cycle, then background threads can receive significantly more fetch slots. In this paper, we evaluate the ICOUNT.2.N [24] fetch partitioning scheme which chooses up to N instructions for fetch from 2 threads every cycle. Under ICOUNT.2.N with slot prioritization, the foreground thread still gets highest priority for fetch; however, background threads can fetch anytime the foreground thread is unable to consume all N fetch slots on a given cycle. In our SMT processor model, we assume the foreground thread terminates fetching on a given cycle if it encounters a predict-taken branch or if it fetches up to an I-cache block boundary. Under these assumptions, it is rare for the foreground thread to fetch N instructions per cycle, opening up significantly more spare slots for background threads to consume.

Instruction Buffers: Foreground Thread Instruction-Window Partitioning. The combination of mechanisms described in Section 2.2 can easily starve background threads of instruction buffer resources. Since the foreground thread always gets fetch priority under slot prioritization, and since the background thread's allocation of instruction buffer entries is limited under either background thread instruction-window partitioning or background thread flushing, it is possible for the foreground thread to consume all instruction buffer resources. Once this happens, the background thread may rarely get buffer entries even if it is allocated fetch slots.

We propose two solutions for increasing background thread instruction buffer allocation that mirror the mechanisms for transparently allocating instruction buffers presented in Section 2.2. First, just as we limit the maximum ICOUNT value for background threads, we can also limit the maximum ICOUNT value for foreground threads. When the foreground thread reaches this instruction count limit, it is not allowed to consume additional fetch slots until its ICOUNT value drops. We call this scheme *foreground thread instruction-window partitioning*.

By limiting the maximum number of foreground thread instructions in the instruction buffers, we reserve some buffer entries for the background threads. However, similar to background thread instruction-window partitioning, this approach is not completely transparent since it allows background threads to take resources away from the foreground thread. The performance impact can be minimized, though, by choosing a large foreground thread ICOUNT limit.

Instruction Buffers: Foreground Thread Flushing. The second scheme for increasing background thread instruction buffer allocation is to pre-emptively reclaim buffer entries occupied by the foreground thread, and to reallocate them to background threads. While arbitrary flushing of the foreground thread will degrade its performance, the impact can be minimized if flushing is initiated at appropriate times. We initiate *foreground thread flushing* when a cache-missing load instruction from the foreground thread reaches the head of the reorder buffer.[1] During such long-latency memory stalls, the instruction buffer entries occupied by the foreground thread do not contribute to its throughput, so flushing will have minimal impact on performance. After flushing, we temporarily disallow the foreground thread from fetching new instructions, thus permitting background threads to fetch into and use the flushed entries. Then, after some number of cycles, we commence fetching for the foreground thread so that it fully recovers the flushed instructions by the time the memory stall completes.

To avoid degrading foreground thread performance, the number of flushed instructions, F, and the number of cycles we allow for flush recovery, T, must be commensurate with the number of cycles that the cache-missing load remains stalled at the head of the reorder buffer. We call this time the *residual cache-miss latency*, R. If R is large, we can afford to flush more foreground thread instructions since there is more time for recovery, thus freeing a larger number of buffer entries. However, if R is small, we must limit the number of flushed instructions since the recovery time is itself limited. Because we expect R to vary on every cache miss, we rely on hardware to estimate R each time we initiate foreground thread flushing, and then select appropriate F and T values to dynamically control the number of flushed instructions and the timing of flush recovery.

We use a cycle counter for every foreground thread load instruction that suffers a cache miss to estimate R. When a load instruction initially suffers a cache miss, we allocate a cycle counter to the load, clear the counter contents, and increment the counter on every cycle thereafter. When

[1] Similar to background thread flushing, we flush the youngest foreground thread instructions from the tail of the reorder buffer, all corresponding instructions in the integer and floating point instruction queues, and all instructions in the instruction fetch queue belonging to the foreground thread.

Parameter	Value	Parameter	Value
Hardware contexts	4	BTB Size	2048 entries
Issue Width	8	Instruction Window Size	128
Fetch Queue Size	32	Load-Store Queue Size	64
Integer/FP Units	8/8	L1/L2 Cache Size	32K/512K
Integer Latency	1 cycle	L1/L2 Block Size	32/64 bytes
FP Add/Mult/Div Latency	2/4/12 cycles	L1/L2 Associativity	4/4
L1/L2 Hit Time	1/10 cycles	Memory Access Time	122 cycles
Branch Predictor	Combined	2-lev Predictor Size	4096 entries
Bimodal Predictor Size	2048 Entries	Meta Table Size	1024 entries

Table 1: SMT simulator settings used for the experiments.

the cache-missing load reaches the head of the reorder buffer, we compute R by subtracting the counter's value from the main memory miss penalty. In Section 3, we will discuss the choice of F and T values as a function of R.

3 Simulation Framework

Our simulation framework is based on the SMT simulator from [13]. This simulator uses the out-of-order processor model from SimpleScalar v2.0, augmented to simulate an SMT pipeline. To evaluate transparent threads, we extended this basic SMT simulator to model the mechanisms presented in Section 2, namely the two mechanisms for sharing instruction slots (slot prioritization and fetch partitioning) and the four mechanisms for sharing instruction buffers (background and foreground thread instruction window partitioning, and background and foreground thread flushing). Table 1 reports the simulator settings we use in our experiments. These settings model a 4x8-way issue SMT processor with 32-entry integer and floating point instruction queues and a 128-entry reorder buffer.

The fetch partitioning model in our simulator considers both the ICOUNT.1.8 and ICOUNT.2.8 schemes, as shown in Table 1. ICOUNT.2.8 requires fetching 16 instructions from 2 threads (8 from each thread) on every cycle [24], and using slot prioritization to select 8 instructions out of the 16 fetched instructions. To provide the fetch bandwidth necessary, our I-cache model simulates 8 interleaved banks, and accounts for all bank conflicts. For the instruction window partitioning schemes, Table 1 shows we assume a maximum background and foreground thread instruction count limit of 32 and 112 instructions, respectively.

As described in Section 2.3, our foreground thread flushing mechanism dynamically selects the number of instructions to flush, F, and the number of flush recovery cycles, T, based on the residual

$R < 8$	F=0	T=0
$8 \leq R < 16$	F=8	T=4
$16 \leq R < 32$	F=16	T=8
$32 \leq R$	F=48	T=16

Table 2: Choice of the number of instructions to flush, F, and the number of flush recovery cycles, T, as a function of the residual cache-miss latency, R.

Name	Description	Input	FastFwd	Simulated
Vpr	Integrated circuit computer-aided design	ref	60M	233M
Bzip2	Compression algorithm	ref	22M	126M
Gap	Language and library designed for computing	ref	105M	157M
Gzip	Data compression program	ref	170M	140M
Equake	Simulates seismic wave propagation	ref	18M	1186M
Art	Models characteristics of organic neural processing	ref	20M	71M
Ammp	Intermediate code for a molecular mechanics program	ref	110M	2439M
Irreg	PDE solver for computational fluid dynamics	144K nodes	29M	977M

Table 3: Benchmark summary. The first three columns report the name, a short description, and the inputs used for each benchmark. The last two columns report the number of instructions in the fast forward and simulated regions.

cache-miss latency, R, at the time flushing is initiated. Table 2 reports the F and T values used by our simulator for a range of R values. Finally, as described in Section 2.2, both background and foreground thread flushing require frequent register map checkpointing since flushing occurs to an arbitrary point in the reorder buffer (as compared to flushing to the last mispredicted branch). For maximum flexibility, checkpointing every instruction would be necessary. To reduce hardware cost, however, our simulator models checkpointing every 8th instruction only. When flushing is triggered, we compute the number of instructions to flush as normal, described in Sections 2.2 and 2.3 for background and foreground thread flushing, respectively. Then, we flush to the nearest checkpointed instruction, rounding up when flushing the background thread (more aggressive) and rounding down when flushing the foreground thread (more conservative).

In addition to the hardware specified in Tables 1 and 2, our simulator also provides ISA support for multithreading. We assume support for a fork instruction that sets the program counter of a remote context and then activates the context. We also assume support for suspend and resume instructions. Both instructions execute in 1 cycle; however, suspend causes a pipeline flush of all instructions belonging to the suspended context. Finally, we assume support for a kill instruction that terminates the thread running in a specified context ID. Our multithreading ISA support is used extensively for performing Transparent Software Prefetching, described later in Section 5.

To drive our simulation study, we use the 7 benchmarks listed in Table 3. The first 3 benchmarks belong to the SPECInt CPU2000 suite, the next 3 belong to the SPECfp CPU2000 suite, and the last is an iterative PDE solver for computational fluid dynamics problems. In all our experiments, we use functional simulation to fast forward past the benchmark's initialization code before turning on detailed simulation. The size of the fast forward and simulated regions are reported in the last two columns of Table 3.

4 Evaluating Transparent Threading Mechanisms

Our experimental evaluation of transparent threads consists of two major parts. First, in this section, we characterize the performance of our transparent threading mechanisms. Then, in Section 5, we investigate using transparent threads to perform software data prefetching.

4.1 Methodology

This section characterizes the performance of our transparent threading mechanisms by studying them in the context of multiprogramming. We perform several multiprogramming experiments, each consisting of 2 - 4 benchmarks running simultaneously on our SMT simulator. A single benchmark from the workload is selected to run as a foreground thread, while all other benchmarks run as background threads. From these experiments, we observe the degree to which our mechanisms maintain background thread transparency (Section 4.2) as well as the ability of our mechanisms to increase transparent thread throughput (Section 4.3).

From the 8 applications listed in Table 3, we use the first 5 for our multiprogramming experiments, grouping benchmarks together based on resource usage characteristics. Of particular significance is a benchmark's *reorder buffer occupancy.* Benchmarks with high reorder buffer occupancy (typically caused by frequent long-latency cache misses) use more instruction buffer resources, whereas benchmarks with low reorder buffer occupancy use fewer instruction buffer resources. Among the 5 benchmarks we use, BZIP2 and ART have high occupancy, EQUAKE and GZIP have low occupancy, while VPR has medium occupancy. In order to stress our mechanisms and to study their behavior under diverse workload characteristics, we group together benchmarks that exhibit both high and low reorder buffer occupancy, using both types as foreground and background threads.

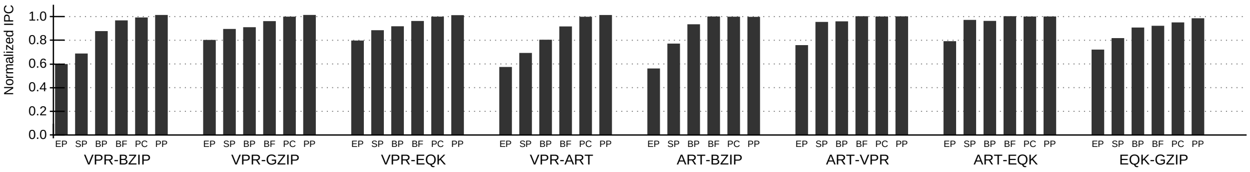

Figure 2: Normalized IPC of the foreground thread when running simultaneously with a single background thread. The bars represent different transparent sharing mechanisms: equal priority (EP), slot prioritization (SP), background thread instruction window partitioning (BP), background thread flushing (BF), private caches (PC), and private predictors (PP).

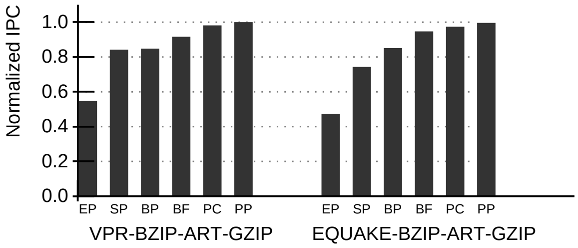

Figure 3: Normalized IPC of the foreground thread when running simultaneously with three background threads. The bars are the same as those in Figure 2.

4.2 Background Thread Transparency

Figures 2 and 3 report the normalized IPC of the foreground thread when running simultaneously with a single background thread and with three background threads, respectively. Groups of bars represent sets of simultaneously running benchmarks, each specified with a label that names the foreground benchmark first followed by the background benchmark(s). Bars within each group represent different transparent sharing mechanisms from Section 2.2 applied incrementally. In particular, the first four bars report normalized IPC with no mechanisms (*i.e.*, all threads have equal priority), with slot prioritization, with background thread instruction window partitioning and slot prioritization, and with background thread flushing and slot prioritization, labeled EP, SP, BP, and BF, respectively. All experiments use the ICOUNT.2.8 fetch partitioning scheme, with all other background thread performance mechanisms disabled. Finally, all bars are normalized to the IPC of the foreground thread running on a dedicated SMT machine (without any background threads).

Figures 2 and 3 show background thread flushing with slot prioritization (BF bars) is the most effective combination of transparent sharing mechanisms. With these mechanisms, the foreground thread achieves 97% of its single-thread performance averaged across the 8 benchmark pairs,

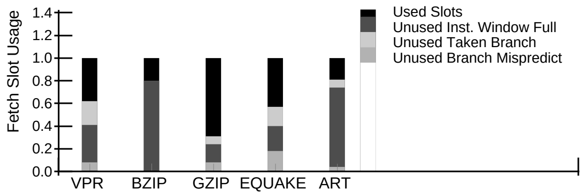

Figure 4: Fetch slot usage for our benchmarks when each benchmark is run on a dedicated SMT processor.

compared to only 70% of single-thread performance when pairs of benchmarks are run with equal priority (EP bars). Background thread instruction window partitioning with slot prioritization (BP bars) also provide good transparency, with the foreground thread achieving 91% of its single-thread performance; however, our results show BP is less effective than BF in all cases. Slot prioritization alone (SP bars) is the least effective, allowing the foreground thread to achieve only X% of its single-thread performance.

Having quantified the transparency of our background threads, we now examine the extent to which the foreground thread's performance degradation is due to sharing memories, a type of resource sharing that our mechanisms do not address. In our SMT model, threads share two types of memory structures: caches and branch predictor tables. To isolate the impact of sharing these structures on foreground thread performance, we replicate them, thus removing any contention due to sharing. The last two bars of each group in Figure 2 report the normalized foreground thread IPC assuming the best mechanisms (*i.e.*, those used for the BF bars) when each context has private L1 and L2 caches (PC bars), and when each context has both private caches and a private branch predictor (PP bars). These results show when cache and branch predictor conflicts are removed, the foreground thread achieves 100% of its single-thread performance. We conclude that our mechanisms enable the background threads to use instruction slots and instruction buffers in a completely transparent fashion, and that further improvements in foreground thread performance can only come by addressing memory sharing.

While Figures 2 and 3 quantify the extent to which background threads are transparent, they do not provide insight into how our mechanisms achieve transparency. To address this issue, we first study how our benchmarks use processor resources. Figure 4 illustrates the usage of the fetch stage, a critical SMT resource. In Figure 4, we break down the total available fetch slots into used and unused slots when each benchmark is run on a dedicated SMT processor. Unused slots are further broken down into three categories indicating the cause for the unused slots: a full instruction fetch queue, wasted slots around a taken branch (after the branch on the same cycle and before the target on the next cycle), and recovery from a branch mispredict. (Although I-cache stalls are another cause of unused slots, an insignificant number of unused slots are due to I-cache stalls in our benchmarks).

Figure 4 sheds light on why our transparent threading mechanisms work. First, the "IFQ Full" components indicate the degree to which our benchmarks occupy instruction buffers, showing that BZIP2 and ART have high instruction buffer occupancy. In Figure 2, we see that any workload using these benchmarks as a background thread exhibits poor foreground thread performance under equal priority. When using equal priority, BZIP2 and ART frequently compete for instruction buffer entries with the foreground thread, degrading its performance. Consequently, in these workloads, background thread flushing significantly improves foreground thread performance since flushing reclaims buffer entries, making the foreground thread resilient to background threads with high instruction buffer occupancy. Conversely, Figure 4 shows GZIP and EQUAKE have low instruction buffer occupancy. In Figure 2, we see that any workload using these benchmarks as a background thread exhibits reasonable foreground thread performance under equal priority, and only modest gains due to flushing.

Second, anytime a workload uses a benchmark with a large "IFQ Full" component as a foreground thread, slot prioritization provides a large foreground thread performance gain and background thread flushing becomes less important. In Figure 2, the ART-VPR and ART-EQK (and to some extent, ART-BZIP2) workloads exhibit this effect. When slot prioritization is turned on, ART gets all the fetch slots it requests and thus acquires a large number of instruction buffer entries (due to its high instruction buffer occupancy), resulting in a large performance boost. At the same time, the background thread receives fewer buffer entries, reducing the impact of flushing.

4.3 Transparent Thread Performance

Figure 5 reports the normalized IPC of the background thread using background thread flushing and slot prioritization on the multiprogrammed workloads from Figure 2. Bars within each workload group represent different transparent thread performance mechanisms from Section 2.3 applied incrementally. Specifically, we report normalized IPC with the ICOUNT.1.8 fetch partitioning scheme without and with foreground thread flushing, with the ICOUNT.2.8 fetch partitioning

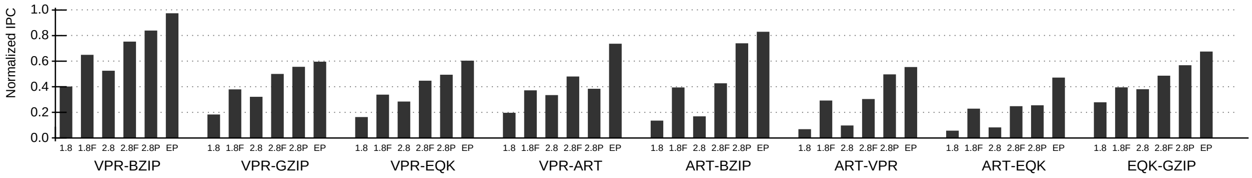

Figure 5: Normalized IPC of the background thread. The bars represent different transparent thread throughput mechanisms: ICOUNT.1.8 without (1.8) and with (1.8F) foreground thread flushing, ICOUNT.2.8 without (2.8) and with (2.8F) foreground thread flushing, ICOUNT.2.8 with foreground thread window partitioning (2.8P), and equal priority (EP). All bars use background thread flushing with slot prioritization.

scheme without and with foreground thread flushing, with the ICOUNT.2.8 scheme and foreground thread instruction window partitioning, and with no mechanisms (*i.e.*, equal priority), labeled 1.8, 1.8F, 2.8, 2.8F, 2.8P, and EP, respectively. All bars are normalized to the IPC of the background thread running on a dedicated SMT machine.

Not surprisingly, the ICOUNT.1.8 fetch partitioning scheme results in the lowest background thread performance, allowing the background thread to achieve only 19% of its single-thread performance on average. Going from ICOUNT.1.8 to ICOUNT.2.8, we see a significant increase in background thread IPC. This is particularly true in workloads where the foreground thread exhibits a large number of "Taken Branch" unused fetch slots (*e.g.*, VPR and EQUAKE as shown in Figure 4) since this is the resource that ICOUNT.2.8 exploits compared to ICOUNT.1.8. In addition, Figure 5 shows foreground thread flushing is important across all workloads, for both ICOUNT.1.8 and ICOUNT.2.8. With foreground thread flushing, the background thread achieves 38% and 46% of its single-thread performance using the ICOUNT.1.8 and ICOUNT.2.8 schemes, respectively. Furthermore, our results show flushing is more important when the foreground thread has a high instruction buffer occupancy (*e.g.*, ART as shown in Figure 4). In these workloads, foreground thread flushing can provide the background thread with significantly more instruction buffer resources, resulting in large performance gains. Interestingly, Figure 5 shows foreground thread window partitioning combined with ICOUNT.2.8 achieves the highest background thread performance, allowing the background thread to achieve 63% of its single-thread performance. Finally, comparing against the EP bars, we see that foreground thread flushing (2.8F) and instruction window partitioning (2.8P) improve the IPC of the background thread to within 22% and X% of the equal priority scheme, respectively.

Although our mechanisms improve background thread performance, it is imperative that they do

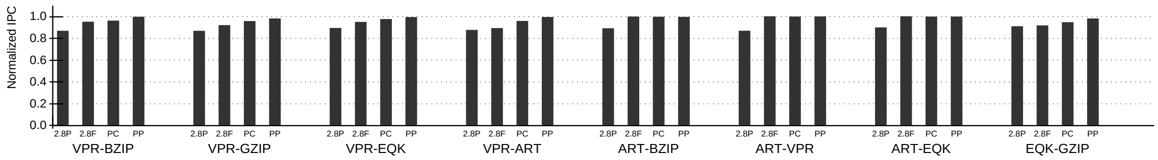

Figure 6: Normalized IPC of the foreground thread. The bars represent different transparent thread throughput mechanisms: foreground thread instruction window partitioning (2.8P), foreground thread flushing (2.8F), private caches (PC), and private predictors (PP). All bars use background thread flushing with slot prioritization.

not sacrifice background thread transparency in the process. Figure 6 plots the normalized IPC of the foreground thread for several of the experiments in Figure 5. This data shows that the increased background thread performance of foreground thread instruction window partitioning compared to foreground thread flushing comes at the expense of reduced foreground thread performance (the 2.8F bars achieve 95% of single-thread performance whereas the 2.8P bars achieve only 84%). We conclude that foreground thread flushing is more desirable since it increases background thread performance without sacrificing transparency. Similar to Figures 2 and 3, the last two bars of Figure 6 remove cache and branch predictor conflicts from the 2.8F bars, showing that practically all of the remaining foreground thread performance degradation is due to memory sharing.

5 Transparent Software Prefetching

This section proposes and evaluates a new sub-ordinate multithreading technique, called *Transparent Software Prefetching* (TSP). TSP performs software data prefetching by instrumenting the prefetch code in a separate *prefetch thread* rather than inlining it into the main computation code, as is done in conventional software prefetching [14, 9, 4]. Prefetch threads run as background threads, prefetching on behalf of the computation thread which runs as a foreground thread. Because they run transparently, prefetch threads incur near-zero overhead, and thus never degrade the computation thread's performance.

TSP solves a classic problem associated with software prefetching: determining what to prefetch. Since software prefetching incurs runtime overhead, it is important to instrument prefetching only for load instructions that suffer a sufficiently large memory access latency so that the benefit of prefetching outweighs the cost of executing the instrumentation code. Identifying the loads for which prefetching is profitable typically requires gathering detailed cache-miss profiles (*e.g.*, summary [1] or correlation [15] profiles). Unfortunately, such profiles are cumbersome to acquire,

(a) Main Thread

```
1 resumeID = LOOP1;
2 resumeContext(cxt_ID);
3   for(i = 0 to N) {
4     /*original loop code*/
5     sync_code;
    }
```

(b) Prefetch Thread

```
  /*dispatcher loop*/
1 void DISPATCHER(/*parameters*) {
2     while(1) {
3       (resumeID)();
4       suspendContext(cxt_ID)  ;
      }
  }

  /*prefetch loop*/
5 void LOOP1() {
6    /*prefetch code*/
7    sync_code;
  }
```

Figure 7: TSP instrumentation example. (a) Computation thread code. (b) Prefetch thread code with dispatcher loop and prefetch procedure.

and may not accurately reflect memory behavior for arbitrary program inputs. In contrast, *TSP eliminates the need for profiling.* Since transparent sharing mechanisms guarantee prefetch threads never degrade the computation thread's performance, prefetching becomes profitable for *all* loads, regardless of their cache-miss behavior. Consequently, TSP can be applied naively, without ever worrying about the profitability of a transformation.

5.1 Implementation

Instrumenting TSP involves four steps. First, we identify any loop that contains one or more array references or indexed array references as a candidate for prefetching. Figure 7a shows such a loop that we will use as an illustrative example. For each identified loop, we copy the loop header and place it into a separate *prefetch procedure.* Inside the copied loop, we insert prefetch statements for each array and indexed array reference appearing in the original loop body (Figure 7b, lines 5-7).

Second, we insert code into the computation thread to initiate the prefetch thread. The thread initiation code is inserted prior to the loop, causing a prefetch thread to execute the prefetch procedure created for the loop. Since the prefetch initiation code is executed by the computation thread, its overhead is not transparent; hence, it is critical to minimize the amount of inserted code. We use a *recycled thread model* [21] to reduce the cost of thread initiation. In the recycled thread model, a single prefetch thread is created during program startup (we assume the thread is statically assigned to a hardware context that is dedicated to running the prefetch thread). Once created, the prefetch thread enters a dispatch loop and suspends itself (Figure 7b, lines 1-3). To initiate a prefetch thread, the computation thread communicates a PC value through memory, and executes a resume instruction that unblocks the suspended thread (Figure 7a, lines 1-2). The prefetch thread jumps indirect through the PC argument (Figure 7b, line 3), executes the prefetch procedure, and returns to the dispatch loop, suspending itself until the next initiation point (Figure 7b, line 4).

Third, we insert code to pass arguments from the computation thread to the prefetch thread. Any variable used by the prefetch thread that is a local variable in the computation thread must be passed. Communication of arguments is performed through loads and stores to a special argument buffer in memory (Figure 7a, lines x-y and Figure 7b, lines x-y). Similar to the thread initiation code, the computation thread's argument passing code does not execute transparently; however, we find this overhead is typically small since only a few arguments are passed in most cases and the argument buffer normally remains in cache.

Finally, we insert code to synchronize the prefetch thread with the computation thread. Because the prefetch thread executes only non-blocking memory references, it naturally gets ahead of the computation thread. We use a loop-trip counter in memory to keep the prefetch thread from getting too far ahead. Every iteration, the computation thread increments the counter (Figure 7a, line x). Another counter, updated by the prefetch thread, is also incremented every iteration (Figure 7b, line y). The prefetch thread compares the two counters, and continues only if they differ by less than the desired *prefetch distance* [14]; otherwise, the prefetch thread busy-waits (Figure 7b, lines z-a). This approach inserts minimal overhead into the computation code. While the prefetch thread may incur a significant number of busy-wait instructions, these instructions execute transparently.

5.2 Performance Evaluation

In this section, we evaluate the performance of TSP, and compare it against two versions of conventional software prefetching: one that naively instruments prefetching for all load instructions, and one that uses detailed cache-miss profiles to instrument prefetching selectively. For selective software prefetching, we use a predicate to evaluate prefetch profitability, and only instrument those static loads for which the predicate is true:

$$Overhead < L1_{miss\_rate} * L1/L2_{latency} + L2_{miss\_rate} * L1/Mem_{latency} \qquad (1)$$

We assume a per-load prefetch cost of 12 instructions and an IPC of 1.5, yielding a prefetch

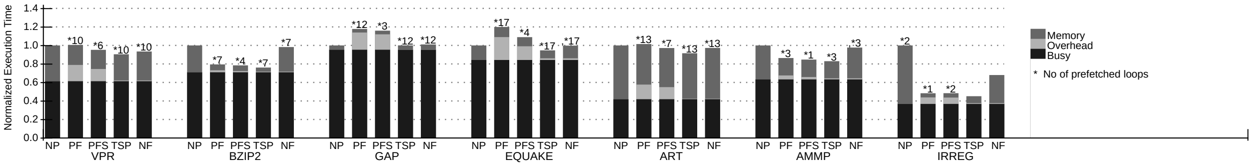

Figure 8: Normalized execution time for different prefetching schemes: no prefetching (NP), naive conventional software prefetching (PF), selective conventional software prefetching (PFS), Transparent Software Prefetching (TSP), and TSP without foreground thread flushing (NF).

overhead of 8 cycles per dynamic load. The L1 and L2 miss rates are acquired by performing cache-miss profiling in each benchmark's simulation region given in Table 3, and we use the L1/L2 and L2/Mem latencies reported in Table 1. Once candidate loads have been selected, we instrument software prefetching by following the well-known algorithm in [14]. Instrumentation for both TSP and conventional software prefetching is performed by hand.

Figure 8 presents performance results for the different prefetching schemes, using 7 benchmarks from Table 3. In Figure 8, we report the normalized execution time for no prefetching (NP), naive software prefetching applied to all candidate loads (PF), selective software prefetching applied to loads meeting our predicate based on detailed profiles (PFS), and TSP applied to all candidate loads. Each bar in the graph is broken down into three components: time spent executing useful instructions, time spent executing prefetch-related instructions, and time spent stalled on data memory accesses, labeled "Busy," "Overhead," and "Memory," respectively. All values are normalized to the NP bars. Finally, a label appears above each bar reporting the number of instrumented loops. These numbers show a significant reduction in loop coverage when performing selective software prefetching.

Our results show TSP outperforms naive conventional software prefetching on every benchmark. Across the 6 SPEC benchmarks, TSP provides a 9.17% performance boost on average, whereas naive conventional software prefetching suffers a 0.8% performance degradation, reducing performance in 4 out of the 6 SPEC benchmarks. This performance discrepancy is due to a 17.8% overhead when using naive software prefetching compared to a 1.16% overhead when using TSP. Despite the fact that prefetching is instrumented for all candidate loads, TSP's negligible overhead enables it to avoid degrading performance even for tough benchmarks like GAP and EQUAKE where there is very little memory stall. Compared to naive software prefetching, selective software prefetching reduces overhead down to 13.4% by using profile information, resulting in a 2.78% performance gain averaged across the 6 SPEC benchmarks. However, TSP still outperforms selective software prefetching on every benchmark. Even for benchmarks where conventional software prefetching performs exceptionally well (*e.g.*, Irreg), TSP still performs better.

The performance gains demonstrated by TSP in Figure 8 suggest that transparent threads not only eliminate overhead, but they also provide enough resources for the prefetch threads to make sufficient forward progress. To evaluate the contribution of our transparent thread throughput mechanisms, the last set of bars in Figure 8, labeled "NF," report the normalized execution time of TSP without foreground thread flushing. The NF bars clearly show the complete set of mechanisms is critical since the performance gains of TSP are significantly reduced when foreground thread flushing is turned off.

6 Related Work

Several researchers have studied hardware resource allocation mechanisms [23, 12, 22] and operating system scheduling policies [19, 18] for SMT processors. In particular, Tullsen and Brown [23] first proposed flushing to reclaim execution resources stalled on long latency memory operations. This work was the motivation behind several of our mechanisms. Compared to these previous techniques, however, our work tries to optimize for single-thread performance rather than overall processor throughput.

Raasch and Reinhardt [16] proposed fetch policies for SMT processors that consider priority in addition to throughput. They assume a single latency-critical foreground thread executes simultaneously with one or more low-priority background threads, and evaluate fetch policies that favor the foreground thread over the background thread(s). While our approach is similar, the critical difference is that our mechanisms permit the background threads to share resources with the foreground thread in a completely transparent fashion. Furthermore, we apply priority mechanisms for slots and buffers along the entire pipeline, rather than just the fetch stage.

Sub-ordinate multithreading is an active area of research. Sub-ordinate threads have been used to execute exception handlers [26] and to improve branch prediction accuracy [5]. More recently, there has been significant interest in using sub-ordinate threads to pre-execute performance-degrading instructions [6, 2, 7, 11, 25, 17]. Our techniques can be used to minimize the impact of sub-ordinate threads on computation thread performance. To demonstrate the potential benefits of transparent threads in this context, we propose and evaluate a new sub-ordinate threading technique that performs software prefetching using transparent threads.

7 Conclusion

This paper investigates resource allocation mechanisms for SMT processors that preserve, as much as possible, the single-thread performance of designated foreground threads, while still allowing background or "transparent" threads to share resources. Our mechanisms ensure transparent threads never take performance-critical resources away from the foreground thread, yet aggressively allocate those resources to transparent threads that do not contribute to foreground thread performance. To demonstrate the potential uses of transparent threads, our work also proposes an implementation of software prefetching on transparent threads, called Transparent Software Prefetching. TSP solves the classic problem of determining what to prefetch. Due to the near-zero overhead of transparent threads, TSP can be applied naively, without ever worrying about the profitability of a transformation.

On a suite of multiprogramming workloads, our results show transparent threads introduce a 3% foreground thread performance degradation on average, and when contention on cache and branch predictor resources are factored out, the performance degradation is less than 1% for all workloads. At the same time, transparent threads run only 22% slower compared to an equal priority scheme. In our evaluation of Transparent Software Prefetching, our results show TSP achieves a 9.17% performance gain across 6 SPEC benchmarks, whereas conventional software prefetching degrades performance by 0.8%. Even when detailed cache-miss profiles are used to guide instrumentation selectively, conventional software prefetching only achieves a 2.78% performance gain. The performance advantage of TSP comes from its 1.16% overhead, compared to a 13.4% overhead for selective software prefetching.

Based on our preliminary results, we conclude that applications running on out-of-order superscalar cores leave a significant number of unused resources that can be allocated to non-critical computations in a completely non-intrusive fashion. We believe our work has only begun to look at the potential uses for such free execution bandwidth. In future work, we hope to further explore the applications of transparent threads, including multiprogrammed workload scheduling, sub-ordinate multithreading optimization, and on-line performance monitoring, as eluded to in this paper.

8 Acknowledgements

The authors would like to thank Dongkeun Kim for contributing to the simulator development effort. Also, the authors would like to thank the anonymous referees for their constructive comments on earlier drafts of this paper.

References

[1] Santosh G. Abraham, Rabin A. Sugumar, B. R. Rau, and Rajiv Gupta. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152. IEEE/ACM, 1993.

[2] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001. ACM.

[3] Thomas Ball and James R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Paris, France, December 1996. IEEE.

[4] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[5] Robert S. Chappell, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *In Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, Atlanta, GA, May 1999. ACM.

[6] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic Speculative Precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, Austin, TX, December 2001.

[7] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001.

[8] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[9] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991. ACM.

[10] James R. Larus and eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[11] Chi-Keung Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001. ACM.

[12] Kun Luo, Manoj Franklin, Shubhendu S. Mukherjee, and Andre Seznec. Boosting SMT Performance by Speculation Control. In *Proceedings of the International Parallel and Distributed Processing Symposium*, San Francisco, CA, April 2001.

[13] Dominik Madon, Eduardo Sanchez, and Stefan Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *Proceedings of EuroPar '99*, pages 716–726, Toulouse, France, August 1999. Springer-Verlag.

[14] Todd Mowry, Monica Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, Massachusetts, October 1992. ACM.

[15] Todd C. Mowry and Chi-Keung Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*. IEEE/ACM, December 1997.

[16] Steven E. Raasch and Steven K. Reinhardt. Applications of Thread Prioritization in SMT Processors. In *Proceedings of the 1999 Multithreaded Execution, Architecture, and Compilation Workshop*, January 1999.

[17] Amir Roth and Gurindar S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pages 191–202, January 2001.

[18] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[19] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2002.

[20] Amitabh Srivastava and alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[21] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Architectural Support for Thread-Level Data Speculation. CMU-CS 97-188, Carnegie Mellon University, November 1997.

[22] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995. ACM.

[23] Dean M. Tullsen and Jeffery A. Brown. Handling Long-latency Loads in a Simultaneous Multithreading Processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, Austin, TX, December 2001.

[24] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 1996 International Symposium on Computer Architecture*, Philadelphia, May 1996.

[25] Craig Zilles and Gurindar Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001.

[26] Craig B. Zilles, Joel S. Emer, and Gurindar S. Sohi. The Use of Multithreading for Exception Handling. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 219–229, November 1999.