Simultaneous Subordinate Microthreading (SSMT)

Robert S. Chappell† Jared Stark† Sangwook P. Kim†‡ Steven K. Reinhardt† Yale N. Patt†

†EECS Department (ACAL)
The University of Michigan
Ann Arbor, Michigan 48109-2122
{robc, starkj, stever, patt}@eecs.umich.edu

‡Intel Corporation
Santa Clara, CA 95052
swkim@mipos2.intel.com

Abstract

Current work in Simultaneous Multithreading provides little benefit to programs that aren't partitioned into threads. We propose Simultaneous Subordinate Microthreading (SSMT) to correct this by spawning subordinate threads that perform optimizations on behalf of the single primary thread. These threads, written in microcode, are issued and executed concurrently with the primary thread. They directly manipulate the microarchitecture to improve the primary thread's branch prediction accuracy, cache hit rate, and prefetch effectiveness. All contribute to the performance of the primary thread. This paper introduces SSMT and discusses its potential to increase performance. We illustrate its usefulness with an SSMT machine that executes subordinate microthreads to improve the branch prediction of the primary thread. We show simulation results for the SPECint95 benchmarks.

1. Introduction

Many current generation microprocessors provide substantial resources in order to exploit Instruction-Level Parallelism (ILP). However, instruction throughput (IPC) in these machines falls well short of ideal. Cache misses, partial issue cycles, branch mispredictions, and insufficient ILP in the program are among the factors preventing full utilization of the available bandwidth. New mechanisms have been proposed to take advantage of this, thereby increasing machine performance. These mechanisms include multithreading and multiple-path execution.

Multithreading improves *overall* performance by simultaneously processing multiple threads. However, multithreading does nothing to improve the performance of each individual thread. When the workload does not provide multiple concurrent threads, this becomes a significant limitation. Multiple-path mechanisms improve performance by simultaneously processing instructions from both paths of conditional branches. However, only a fraction of the work is guaranteed to be useful, since there is only one correct path.

Simultaneous Subordinate Microthreading (SSMT) operates under the same principle as multithreading and multiple-path mechanisms: supply additional useful work to the machine to exploit unused processing bandwidth. A key concept that differentiates SSMT is that the additional work is done to enhance the performance of microarchitectural structures, solely for the benefit of the primary thread.

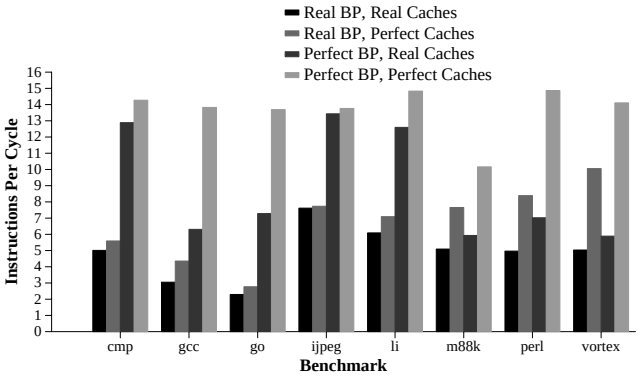

Figure 1. Performance potential of SPECint95 benchmarks.

By improving items critical to performance, such as branch prediction accuracy and cache behavior, we can achieve a boost in performance. Figure 1 compares the performance of a machine with real branch prediction and real caches to that of a theoretical machine with perfect branch prediction and perfect caches (see Section 5.1 for experimental setup). There remains a significant amount of untapped performance potential to be had by improving branch prediction and cache behavior. Therefore, it makes sense to supply the processor with additional work aimed

toward this end. This approach is the basis of Simultaneous Subordinate Microthreading.

This paper describes the SSMT mechanism and discusses and how it can be used to enhance the performance of single-threaded applications. Section 2 describes previous work. Section 3 describes SSMT and how it can be used. Section 4 provides a detailed example of SSMT targeted at improving branch prediction. Section 5 provides simulation results for our branch prediction mechanism. Section 6 provides conclusions.

2. Previous Work

Several studies have examined the notion of increasing machine performance by supplying additional concurrent work to the execution core. Two approaches are multithreading and multiple-path execution.

2.1. Multithreading Mechanisms

A multithreaded machine [15, 13, 7, 17] has the ability to process instructions from several different threads without performing context switches. The processor maintains a list of active threads and dynamically decides which thread's instructions to issue into the machine.

The co-existence of multiple active threads allows a multithreading processor to improve performance by taking advantage of Thread-Level Parallelism (TLP). Instructions from different threads are independent of one another and thus can be executed in parallel, leading to greater functional unit utilization and greater tolerance of execution latencies. Instruction cache misses also can be better tolerated. If one thread suffers a miss, the processor can still fetch instructions from the other threads. The handling of branches similarly benefits from the existence of multiple active threads. Either each thread waits for its branches to be resolved before proceeding, or if branch prediction is used, only the mispredicted thread need be flushed and redirected.

Multithreading has been shown to improve the overall performance of the processor. However, it is important to note that the performance of each individual thread is not improved by multithreading. Furthermore, it is likely that the performance of each thread will be degraded, since resources must be shared among all active threads. In short, multithreading may make sense when running a multiprogrammed or multithreaded workload, but provides no benefit when considering one single-threaded application. Resources required to maintain several active contexts create additional complications. A modified fetch mechanism is necessary, including multiple fetch points, multiple prediction structures, and possibly the ability to issue from more than one thread per cycle.[1] Each thread requires its own register renaming table. The number of physical registers may also need to be increased to maintain the same renaming capability. Available cache memory must be tagged and shared by the active threads.

2.2. Multiple-Path Mechanisms

Mechanisms for multiple-path execution have been proposed [19, 4, 20]. These implementations can vary widely, but the fundamental approach is the same. Branch mispredictions are a major performance limitation. Rather than predicting all of the conditional branches in a program, a multiple-path machine issues instructions from both taken and not-taken paths. Later, when the branch is resolved, the incorrect path is flushed from the machine.

Multiple-path machines appear attractive because mispredictions are removed by executing both paths of a branch. The two paths of a branch are independent, so they execute in parallel, if there are sufficient resources.

One problem with multiple-path mechanisms is that some of the work done by the machine is guaranteed to be thrown away after the branch is resolved. Instructions on incorrect paths cause greater contention for resources. Most notably, wrong-path memory references (instruction and data) place increased demand on the caches and the memory system that can lead to additional latency for the correct path. Deeply speculating down multiple paths exacerbates this problem. Thus, the performance improvement of multiple-path mechanisms is contingent upon the benefit of fewer mispredictions outweighing the penalties due to fetching and executing wrong-path instructions.

3. SSMT

3.1. Overview

This section describes the SSMT mechanism and the various changes needed to implement it in a modern dynamically-scheduled processor. As discussed above, the goal of SSMT is to use additional threads to improve the performance of a single primary thread. These additional threads are supplied in the form of microcode, and we refer to them as *microthreads*.

Figure 2 illustrates an SSMT machine. Microthread routines are code sequences written in an internal instruction format of the processor. They can be stored on-chip

[1] Multithreading originally fetched and executed instructions in-order from multiple threads [15]. Simultaneous multithreading first meant fetching instructions from more than one active thread in a single cycle and executing them out-of-order using the same core [18]. A subsequent improvement restricted fetch to one thread per cycle, but still allowed concurrent execution of instructions from multiple threads [17]. This taxonomy was first noted in [14].

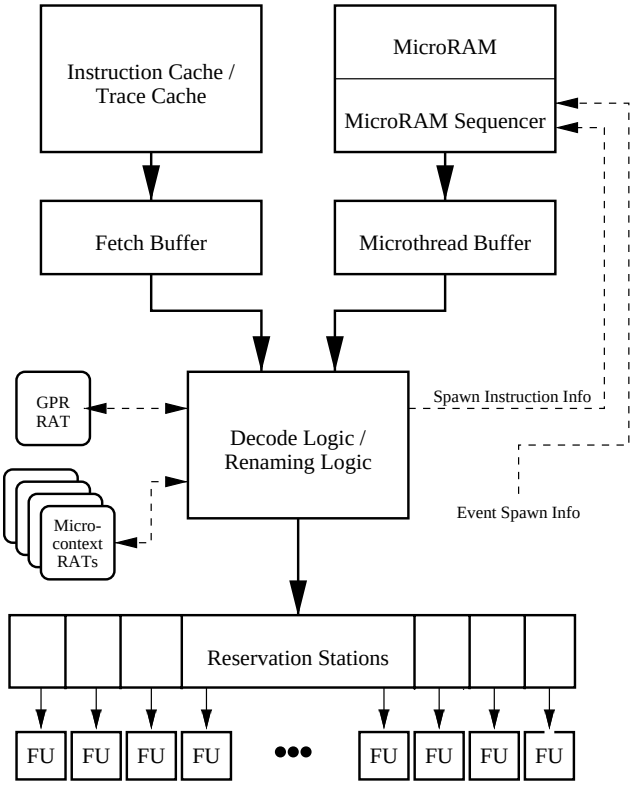

Figure 2. High-level block diagram of an SSMT machine.

in a microcode RAM (the *microRAM*). When certain processor events occur, a microthread is *spawned*, and corresponding microthread instructions are injected into the decode/rename stage. These instructions are renamed separately from those from the primary thread by using separate Register Alias Tables (RATs). The issue logic can then select some combination of primary-thread and microthread instructions to issue to the machine's reservation stations. These instructions can be executed simultaneously by the machine's out-of-order execution core.

An SSMT machine provides two distinct methods for spawning microthreads. The first, the SPAWN instruction, is part of the Instruction Set Architecture (ISA). The encoding of this instruction specifies a microthread routine, which is injected as a result of SPAWN instruction decode (more detail in Section 3.2.1). The second method, an *event spawn*, is triggered by an instance of one of a pre-defined set of events that may occur during execution of the primary thread. Each type of event spawns a different microthread, much the same way interrupts invoke different handlers. Examples include branch mispredictions, cache misses, and timer expirations.

3.1.1. Advantages of SSMT. Microthread optimizations have several advantages over purely hardware-based mechanisms. First, sophisticated algorithms can be written in microcode without concern for hardware complexity. Microcode instructions are executed using the existing datapath. Second, microthread optimizations are very flexible. Routines can be tuned to specific applications and processor implementations, or even disabled altogether when unnecessary.

An alternative to microthread optimization is to provide similar routines as separate conventional threads of execution. However, a microthreaded approach has several advantages. Since optimization routines are small and serve a specific purpose, it makes sense to manage them separately from the primary thread using the microRAM. This prevents the fetch of microthread instructions from ever impacting the fetch of primary thread instructions. Furthermore, microthreads are not stored in the instruction cache, and thus never suffer cache misses. This means that microthread instructions are available for issue even when primary thread instructions aren't due to a cache miss. In addition to these benefits, microthreads are written in an internal instruction set. This allows specialized instructions to be used without amending the ISA. This likely requires some microthreads to be rewritten for each new processor implementation. However, in order to achieve the highest performance, microthreads should always be hand-optimized to each implementation.

This framework makes it possible to create powerful microthread routines to perform various optimizations on behalf of the primary thread. Note that it is certainly possible to include several different microthread enhancements for a single application. More details on microthread optimizations, as well as some examples, are given in Section 3.3.

3.2. Support for SSMT

Several modifications are necessary to implement Simultaneous Subordinate Microthreading in a traditional superscalar out-of-order machine.

3.2.1. ISA Support. Implementation of SSMT requires additions to the ISA. New instructions to load and unload the microRAM structure are necessary. For example, microthreads could be loaded with specialized store instructions that write to addresses within microRAM. Alternatively, a single instruction could be added to copy a portion of main memory to the microRAM.

The SPAWN instruction was briefly mentioned in Section 3.1. This instruction is used to explicitly spawn microthreads from the primary thread. The encoding of SPAWN is as follows:

opcode	*microthread number*	*parameter*

The sizes of these fields depend on the ISA. We have assumed a 6-bit *opcode*, a 6-bit *microthread number*, and a 20-bit *parameter* field. The *microthread number* corresponds to a specific microthread routine stored in the microRAM. When a SPAWN instruction is decoded, the microRAM is accessed with the microthread number, and the corresponding routine injected into the machine. The *parameter* field contains a literal value that is passed to the microthread routine via a Microcode Special Purpose Register (MSPR).

In order for a microthread to operate independently of the primary thread, it needs to maintain separate architectural state, most notably a separate set of registers. We call the state unique to each microthread invocation a *microcontext*. A new microcontext is created when a microthread is spawned. It consists of a set of Microthread General Purpose Registers (MGPRs) and a set of Microthread Special Purpose Registers (MSPRs). The MGPRs are simply general purpose registers used by an individual microthread. Our implementation assumes MGPRs cannot be used to communicate values between different microthread invocations. The MSPRs are read-only registers that contain information about the state of the primary thread that existed when the microthread was spawned. These values can be used by microthreads to guide optimizations.

3.2.2. Compiler and OS Support. SSMT requires support from the compiler. We expect that the microthread routines will be written and optimized by hand to achieve maximum performance. However, an SSMT compiler is important for analyzing the behavior of the primary thread and choosing microthread optimizations to include. For example, a compiler might use a profiling run of a benchmark to speculate if low branch prediction accuracy is a major limitation. If so, the compiler could choose to include an appropriate microthread optimization. In addition to microthread selection, other aspects of the microthread mechanism require support from the compiler and/or operating system. These include microRAM management, microthread memory allocation, and microthread data initialization.

In order to be used, microthread routines need to be loaded into the microRAM before execution of the program. Our implementation assumes that microthread routines are linked into the executable image of the program. To load them into the microRAM, the compiler simply inserts instructions into the program's startup code. Since we expect practical microthread routines to be small (well under 100 instructions each), this would not significantly increase the size of the program executable.

Part of the power of microthreads comes from their ability to use processor main memory through the load/store datapath. In our mechanism, we use the compiler to map memory space in the global data segment for this purpose. The global pointer offsets associated with load/store instructions in the microthread routines need to be adjusted for each application. Thus, the additional memory needed by microthreads appears as part of the primary thread, though no primary thread instructions ever access it. Note that it is also possible for microthreads to use dedicated on-chip "scratch-pad" memory or memory space completely managed by the operating system.

Some microthreads use data structures that require initialization. We assume that the program startup code is responsible for this, in addition to loading the microRAM.

Our current implementation assumes much of the support for SSMT is provided by the compiler. However, some operating system support is still necessary. As mentioned above, microthread routine numbers are encoded in the SPAWN instructions, and program specific memory offsets are encoded in the microthread load/store instructions. As such, the contents of the microRAM must be considered as part of the context of the primary thread. To support this, the operating system could simply flush and reload the microRAM during context switches. Alternately, a more aggressive implementation might include microRAM entries tagged with context identifiers or multiple microRAM context "windows".

It is important to note that this section describes support for microthread routines that is somewhat tied to the program being optimized. It is also easy to imagine more general microthread routines that are not tied to any particular program, and do not necessitate this kind of support. It makes more sense to divorce such generalized routines from the program's context and to support them entirely within the operating system.

3.2.3. Hardware Support. A major advantage of Simultaneous Subordinate Microthreading is that it builds upon hardware that is already present in the processor. This section describes the additional hardware needed to support the general SSMT mechanism.

A key component of our SSMT mechanism is the microRAM. The number of microthread routines that the microRAM can contain is governed by the size of the *microthread number* field of the SPAWN instruction. The microRAM can be accessed by microthread number (to support the SPAWN instruction) or by microthread instruction number (to support branches within the microthread routines). Many processors already incorporate micro-engines with similar capabilities in their designs. It may be possible to leverage these existing micro-engines to implement the functionality of the microRAM.

Since microthread instructions are stored on-chip, no modification of the primary *fetch* mechanism is necessary to support microthread injection. However, the issue mechanism (decode/rename) must support issue from both the primary thread and microthreads. As suggested in Section

3.1.1, an aggressive implementation would likely issue from both simultaneously.

As noted in Section 3.2.1, a microcontext is created each time a microthread is spawned. The hardware must support at least one active microcontext in order to execute a microthread concurrently with the primary thread. Adding hardware to support multiple active microcontexts allows additional microthread routines to be executed concurrently. If a spawn is encountered when no hardware microcontexts are available, it can be queued until one is available, or ignored altogether.

Each microcontext includes additional register sets, as described in Section 3.2.1. MGPRs require expansion of the register renaming logic. In order to maintain the same renaming capability, the number of physical registers should be increased proportionally to the number of MGPRs and supported hardware microcontexts. We assume 8 MGPRs per microcontext in our implementation. MSPRs can be directly sourced by microthread instructions, since they are never written within a microthread. The number of MSPRs needed is determined by the information that a processor implementation makes available to the microthreads.

3.3. Using SSMT to Enhance Performance

SSMT is a general mechanism that can be applied in many ways to increase performance. This section concentrates on three areas that have potential for improvement with SSMT: branch prediction, prefetching, and cache management.

3.3.1. Branch Prediction. Studies have examined use of the compiler to "synthesize" dynamic branch prediction [11, 1]. By analyzing program semantics, the compiler can insert instructions in the program to communicate with the processor's branch prediction hardware. In many cases, this can lead to very accurate prediction for certain types of branches. For example, general branch prediction schemes are often unable to capture enough history to accurately predict FOR-style loop exits, even though these are conceptually easy to predict at run-time. In these cases, the compiler could synthesize a prediction by inserting instructions to communicate the iteration count to the branch prediction hardware.

An SSMT machine provides an excellent opportunity to enhance compiler-synthesized branch prediction by using microthreads to do all the work. Synthesis routines would be stored on-chip and invoked with a SPAWN instruction. This limits the code bloat from the synthesized predictions to a single instruction. Furthermore, an SSMT microthread could communicate with the branch prediction hardware without external ISA modifications. It is easy to imagine a microthread routine that would implement the loop predictor described in the previous paragraph.

Though SSMT provides some useful enhancements to existing compiler-synthesized branch prediction, the power and flexibility of microthreads provides opportunity to go well beyond what is reasonably compiler-synthesized. For example, a complete branch predictor can easily be implemented by a microthread routine. Such a microthread predictor could be used as an alternative to the processor's hardware branch predictor for some branches. This is the basis of the detailed SSMT example provided in Section 4.

It is important to reiterate that SSMT microthread enhancements are not mutually exclusive. One could easily envision combining multiple branch prediction enhancements. As mentioned above, any combination of microthread routines can be selected to maximize performance of the primary thread.

3.3.2. Prefetching. Prefetching is an important mechanism for improving memory system performance. Many software-based and hardware-based mechanisms have been proposed [9, 2, 3, 10]. An SSMT machine provides opportunity for creating extremely sophisticated prefetches.

Most existing ISAs that support prefetching provide an instruction, such as Alpha's FETCH, to load a single cache line into the first-level cache. In an SSMT machine, the compiler can insert a single SPAWN instruction into the program that invokes an entire prefetching *routine*. Such a routine could prefetch several disjoint cache lines at once, prefetch an entire array using a microthread loop, or even prefetch pointer-linked structures by performing a traversal within the microthread. Routines could also be constructed to *conditionally* prefetch based on information gathered at run-time. As with all microthread optimizations, prefetching routines can be processed concurrently with instructions from the primary thread.

An SSMT machine could also implement many existing hardware prefetching algorithms by using event spawns. For example, shadow-directory prefetching [3] operates by tracking cache miss patterns and using this information to later prefetch chains of cache lines. Using a microthread spawned by cache miss events, a more sophisticated version of this general algorithm could be implemented *without any additional hardware*. Further prefetching enhancement can be gained by using the SSMT mechanism to provide feedback to the prefetching algorithm. Microthreads can be spawned on events such as detecting a useful prefetch, useless prefetch, or late prefetch. These feedback microthreads could alter the prefetching algorithm to prioritize useful prefetches, filter out useless prefetches, and perhaps reschedule late prefetches. This idea is similar in approach to that of informing memory operations [8], but has all of the advantages associated with using microthreads.

3.3.3. Cache Management. Caches use replacements policies to decide if a line should be replaced, and if so, which line should be replaced. Most caches always replace a line on a miss, although that need not be the case. Set associative caches typically approximate some form of Least Recently Used replacement to decide which line in a set to replace. In any case, reasonable replacement policies are limited to simple state machines that are easily embedded in cache structures. Using microthreads, it is possible to implement a more intelligent replacement algorithm without adding hardware complexity. For example, consider a microthread that is spawned on every cache miss. This microthread could maintain cache miss information about every cache line touched by the primary thread, and thus provide feedback to guide the replacement policy of the cache. For example, frequently-used lines that are constantly displaced could be locked into the cache to prevent further misses. Thrashing situations could be identified and possibly remedied by re-mapping conflicting lines. Alternatively, a microthread could manage the contents of a victim cache to achieve a similar goal.

Used in concert with microthreads that manage prefetching strategies, cache management can be expanded to include microthread management of all levels of the cache hierarchy. In the future, we expect a cache to consist of three parts: a tag store, a data store, and a microthread cache management routine.

4. An SSMT Example: Branch Prediction

It is well-known that accurate branch prediction is required to achieve high performance in a wide-issue, deeply-pipelined processor. Most current generation processors devote a significant portion of hardware to implement sophisticated branch prediction algorithms. However, even for the most advanced prediction schemes, such as the multi-hybrid [5] and the variable length path-based predictor [16], a significant fraction of performance is lost due to branch mispredictions, as was shown in Figure 1. In this section, we present a branch prediction mechanism that is enhanced through the use of a microthread-based branch predictor. This example demonstrates the potential usefulness of SSMT. It is not an attempt to build the ultimate branch predictor.

Predictors implemented with microthreads can be larger and more complex than predictors implemented in hardware. However, hardware predictors generate predictions faster than microthread predictors and are very accurate for some branches. Hence, we use a complex microthread-based predictor only for those branches that are likely to be poorly predicted by the hardware predictor. Conceptually, this is the same idea that makes a hybrid predictor [12] more accurate than any of its component predictors. In our case, the "hybrid" we are building includes two components: the hardware predictor of the processor, and the microthread-based predictor supplied via the SSMT mechanism.

We use profiling to identify conditional branches suited to microthread prediction. For each branch, our profiling algorithm estimates the performance to be gained or lost by predicting it with the microthread predictor. The heuristic we use is described in Section 5. Branches that show estimated gains are predicted with the microthread predictor.

Once the branches that will be predicted via microthread are identified, the compiler inserts a SPAWN instruction after each to invoke the microthread branch predictor. At run-time, the SPAWN instruction invokes the microthread to generate the prediction that will be used the *next* time the branch is fetched. The prediction for the current instance of the branch was generated by the *last* invocation of the microthread. The microthread stores its prediction into a *prediction cache*, along with the branch's fetch address. Both the prediction cache and the hardware predictor are accessed at fetch time. Fetches that hit in the prediction cache use the stored prediction and then invalidate it. Otherwise, the hardware prediction is used.

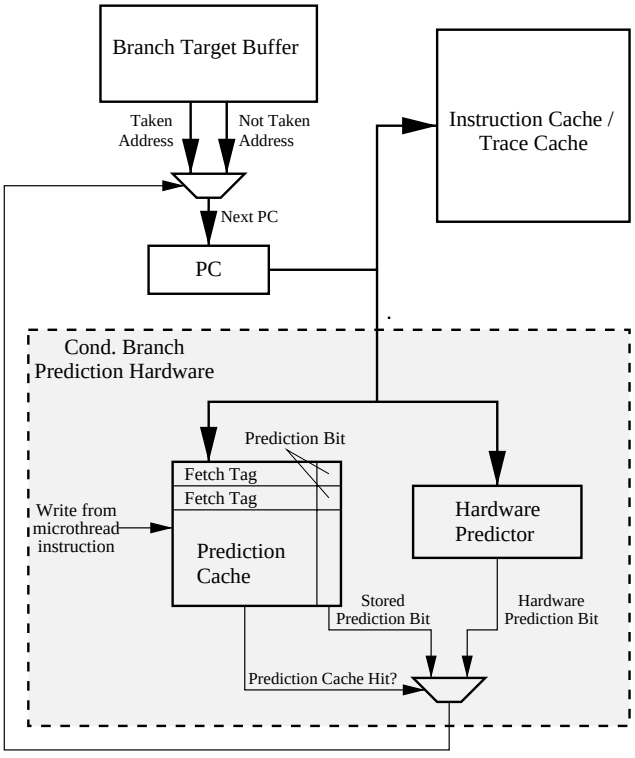

Figure 3. Hardware needed to support the microthread branch prediction mechanism (modifications shown in the shaded area).

The hardware necessary to support this branch prediction

mechanism is shown in Figure 3. Hardware added to support the prediction cache mechanism is shown in the shaded region. It should be noted that the prediction cache is a general mechanism to influence the processor's branch prediction behavior. If several different microthread branch predictors are implemented, they all use the same prediction cache. In other words, the prediction cache is not tailored to any specific microthread routine.

Using this mechanism, we achieve higher prediction accuracy than if the processor's hardware predictor were used alone. However, since every execution of a spawn instruction creates additional work for the machine, we must be careful which branches spawn microthreads. If the time saved by removing mispredictions is greater than the time spent executing the additional microthread instructions, then our mechanism will be successful. Our results are presented in the next section.

5. Experiments

This section presents additional implementation details and experimental results for the SSMT prediction example described in the previous section.

5.1. Experimental Setup

We modeled machines that can fetch, execute, and retire up to 16 instructions per cycle. The window size was 512 instructions. The instruction and data caches were 64 KBytes (KB), direct mapped, with a 64 byte line size, and a 10 cycle miss penalty. We modeled two different conditional branch predictors. The first predictor was a 16 KB gshare [12] predictor. The second predictor was a hybrid consisting of an 8 KB variable-length path (VLP) predictor [16] and an 8 KB SAg predictor [21]. The minimum branch misprediction penalty was 7 cycles.

A processor with a poor fetch mechanism will underutilize its execution resources. In such an environment, microthread instructions would rarely compete with primary thread instructions for execution resources. To prevent biasing of our results due to this, we modeled a somewhat idealized fetch mechanism for the primary thread. Our machine is capable of handling any number of branches per cycle, possibly sourcing multiple lines from the instruction cache in a single fetch.

We ran our experiments using the 8 SPECint95 benchmarks. All benchmarks were compiled for the Alpha ISA using the Digital compiler with optimizations –O2 and –Olimit 3000. All benchmarks were run to completion.

5.2. The Microthread Predictor

Our baseline hardware predictor is a 16 KB gshare predictor. It accurately predicts branches that exhibit strong global correlation [6]. A weakness of this predictor is its inability to predict branches that only exhibit self (or per-address) correlation. To compensate for this weakness, our SSMT microthread routine implements a PAg [21] predictor. In addition to providing more accurate predictions for some branches, PAg is a simple predictor well-suited to microthread implementation.

A PAg predictor goes through two steps to generate a prediction. The first step is a lookup into the Branch History Table (BHT). The BHT contains one branch history for each static branch that can be predicted by the microthread routine. In our implementation, the index into the BHT is given by the SPAWN instruction's *parameter* field. In the second step, the history from the BHT entry is used to index the Pattern History Table (PHT), which is a table of saturating counters. The high bit of the PHT counter is used to make the prediction for the branch.

Our PAg scheme uses 24-bit local histories and 3-bit PHT counters stored one per byte. Thus, the PHT requires 16MB of storage. An equivalent PHT is unthinkable to implement in hardware, yet can be done simply by a microthread routine using main memory. Note that our predictor, unlike a hardware PAg predictor, never suffers from loss of local history due to misses in a BHT storage structure.

The PAg microthread routine takes 15 instructions to generate a prediction and to update the BHT and PHT. We do not present the routine itself due to lack of space. It is available at `http://www.eecs.umich.edu/HPS`.

Figure 4 shows the performance of conventional machines using both branch predictors (gshare and VLP/SAg hybrid). It also shows the *potential* machine performance when using the microthread PAg predictor in conjunction with each hardware predictor. A machine with perfect branch prediction is shown for reference. This graph shows only potential performance using the microthread PAg predictor—the true performance depends on the impact of injecting the PAg microcode and is discussed in Section 5.4.

5.3. The Compiler Selection Heuristic

The compiler identifies the branches to be predicted with the microthread routine. We created a very simple, profile-based heuristic to make the selection. For each static branch, the compiler compares the prediction accuracy of the hardware predictor to the accuracy of the microthread PAg predictor. If the accuracy of the PAg is worse, that branch will always use the hardware predictor. Otherwise,

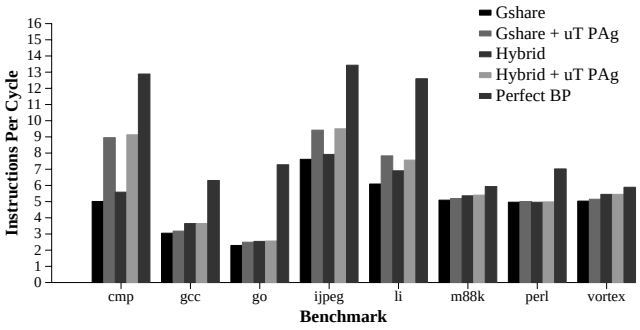

Figure 4. *Potential* **machine performance with and without microthread predictions.**

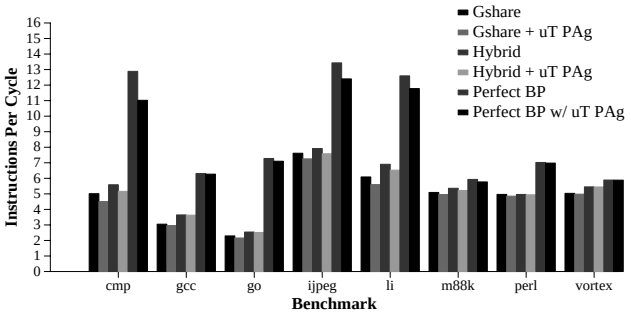

Figure 6. Machine performance with microthread overhead, but without microthread predictions.

the compiler estimates the number of cycles saved using the PAg predictor by multiplying the number of mispredicts eliminated by the average resolution time of the branch. The compiler then estimates the number of cycles lost to executing microcode by multiplying the dynamic execution count of the branch by the number of cycles needed to issue one invocation of the microthread routine. If the compiler computes a net savings in cycles, a spawn instruction is inserted for that branch, and it will use the microthread prediction. If the compiler computes a net loss, the branch will be predicted using the hardware predictor.

5.4. Performance Analysis

Figure 5 shows performance of machines that do and do not use the microthread PAg predictor. For this experiment, the microthread routine is injected to generate the PAg predictions. Five of the eight benchmarks (gcc, go, m88ksim, perl, and vortex) show approximately no change in performance. This follows what we would expect from Figure 4. These benchmarks saw little potential gain from including PAg predictions, and thus we see little actual gain, or even slight loss, when the microcode is injected to generate them.

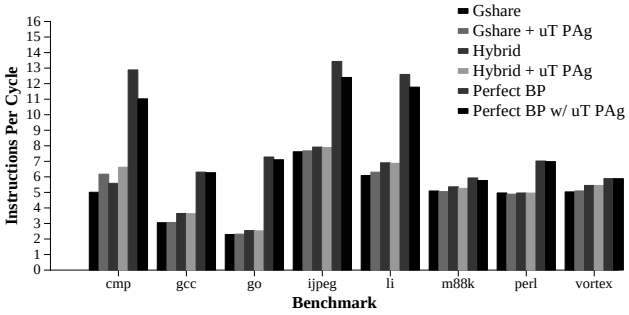

Figure 5. Actual machine performance with and without microthread predictions.

Compress, ijpeg, and li all showed significant potential gains in Figure 4. However, only compress saw a significant actual gain in Figure 5. For ijpeg and li, the overhead of the microthread predictor nullifies the benefit of the increased prediction accuracy. We examine causes for this below.

Figure 6 shows performance of machines that do and do not use the microthread PAg predictor. For this experiment, the microthread routine is injected, but the hardware predictions are always used. Therefore, none of the machines benefit from the microthread PAg predictor. Any performance loss is due solely to microthread overhead. Compress, ijpeg, and li all show significant loss from executing microthread instructions. This can be attributed to three factors:

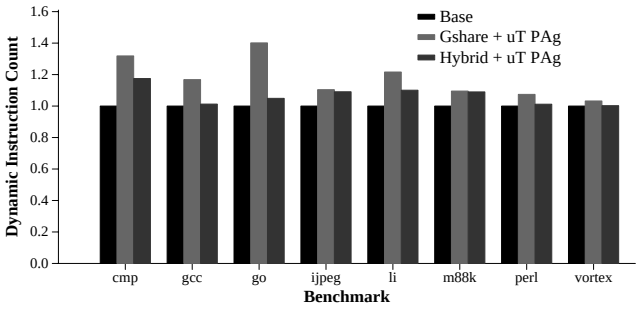

Figure 7. Normalized instruction counts with and without microthread predictions.

- The microthread routine steals some issue and execution bandwidth from the primary thread. Figure 7 shows the dynamic instruction counts for each benchmark normalized to the counts of the machines not using the microthread predictor. It is interesting to note that compress shows a large increase in dynamic instructions (35% for the gshare version), but also shows the greatest performance improvement in lieu of these additional instructions. Go shows an even larger in-

crease in dynamic instructions, but almost breaks even on performance. The same is true to a lesser extent for the other benchmarks. This suggests that, in general, there is capacity to execute the additional microthread instructions and that the majority of the performance loss in Figure 6 can be traced to another source.

- The microthread routine introduces additional memory instructions, which compete for memory bandwidth and cache space. The PAg microthread routine contains 4 memory instructions (2 loads and 2 stores). In addition, it has a memory footprint in excess of 16 megabytes. Both of these factors, competition for bandwidth and cache space, reduce performance of the primary thread. To gauge this impact, we ran the same experiment as in Figure 6, but with perfect data caches. With a real data cache and the gshare predictor, the gap in average performance between configurations using and not using the microthread predictor was .20 IPC. With a perfect data cache, this gap shrinks to .17 IPC, a change of only .03 IPC. The largest change for a single benchmark was only .1 IPC. The small changes in relative performance between experiments using real and perfect data caches suggests that contention in the memory system is not responsible for the loss in performance shown in Figure 6.

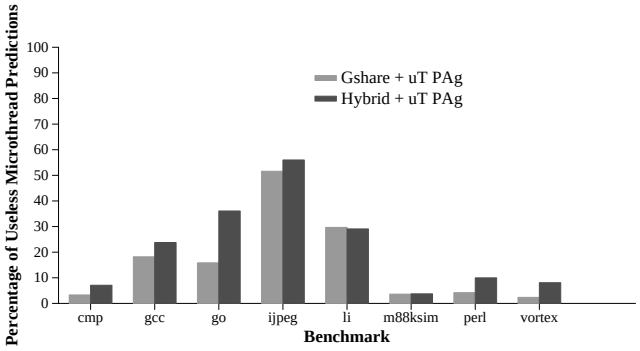

Figure 8. Percentage of useless microthread predictions.

- When the prediction cache is accessed, the cache may not contain the prediction for the associated branch—even if the compiler specified that the branch should be predicted with the microthread PAg predictor. This occurs when the PAg microthread routine doesn't have enough time to compute the prediction and write it into the prediction cache before the prediction is needed; that is, before the branch is encountered again. When there is a miss in the prediction cache, the hardware prediction is used by default.

  In some cases, a microthread prediction is still useful even if it isn't written into the prediction cache on time. Our implementation assumes that a microthread prediction is always more accurate than a hardware prediction. If a microthread prediction for a branch is not known at fetch, but is computed before the branch resolves, the microthread prediction is compared to the hardware prediction that was used. If the predictions are different, the microthread prediction overturns the hardware prediction, and fetch is redirected according to the microthread prediction.

  In other cases, the microthread prediction is not computed before the branch resolves, so the prediction is completely useless. Figure 8 shows the percentage of useless predictions. The impact this has on performance depends on how well the hardware predicts these branches. For compress, the percentage of useless predictions is low, so the overall prediction accuracy is not greatly affected. For li and ijpeg, whose *potential* performance gains were large (see Figure 4), this impact is certainly harmful, and probably accounts for the lack of any *actual* gains.

Our PAg microthread predictor could be improved in several ways. Our experimental data suggests that the profile-based heuristic was overly optimistic in selecting branches for microthread prediction. In many cases, the cost of performing a microthread prediction for a branch outweighed its benefit. A better heuristic would better identify these cases, and prevent those branches from being selected for microthread prediction. Furthermore, useless microthread predictions limited the performance gains in two of the benchmarks (li and ijpeg). A better heuristic would disable microthread predictions for branches that execute in rapid succession, as these predictions probably can't be generated in time for them to be useful. The branch prediction hardware might also monitor which SPAWN instructions generate useless predictions. The hardware could then disable the injection of the microcode routines for those SPAWN instructions deemed not useful.

6. Conclusions

Current multithreading paradigms do not improve the performance of single-threaded applications. To remedy this, we proposed a new multithreading mechanism, Simultaneous Subordinate Microthreading (SSMT), which uses a machine's otherwise unused execution resources to boost the performance of a single primary thread. To accomplish this, SSMT dynamically injects microcode optimization threads that manipulate the microarchitecture to enhance performance-critical mechanisms. We described the

implementation of the general SSMT mechanism, and several possible SSMT applications. To demonstrate its usefulness, we provided a simple example of how it can be used to increase branch prediction accuracy. Though the example was limited in scope, it demonstrated the usefulness of SSMT by showing that significant performance gains are possible using the mechanism.

Acknowledgments

We greatly appreciate the support of our industrial parters: Intel Corporation, HAL Computer Systems, and Advanced Micro Devices. In addition, we especially thank Dan Friendly, Paul Racunas, and the other members of our research group for their insights and comments on drafts of this paper.

References

[1] D. August, D. Connors, J. Gyllenhaal, and W. Hwu. Support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Proceedings of the Third IEEE International Symposium on High Performance Computer Architecture*, pages 84–93, Feb. 1997.

[2] D. Bernstein, D. Cohen, A. Freund, and D. Maydan. Compiler techniques for data prefetching on the powerpc. In *Proceedings of the 1995 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1995.

[3] M. Charney and T. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM Journal of Research and Development*, 41(3):265–286, May 1997.

[4] J. C. Dehnert, P. Y. T. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 26–38, 1989.

[5] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 3 – 11, 1996.

[6] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52 – 61, 1998.

[7] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizwa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[8] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[9] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[10] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[11] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediciton. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec. 1996.

[12] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[13] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 342–351, 1991.

[14] Y. N. Patt. Keynote Address, Workshop on Simultaneous Multithreading (HPCA-4), 1998.

[15] B. J. Smith. A pipelined shared resource mimd computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, 1978.

[16] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch predcition. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 170 – 179, 1998.

[17] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, 1996.

[18] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[19] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 313–325, 1995.

[20] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[21] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, 1992.