Threaded Multiple Path Execution

Steven Wallace Brad Calder Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego
{swallace,calder,tullsen}@cs.ucsd.edu

Abstract

This paper presents Threaded Multi-Path Execution *(TME), which exploits existing hardware on a Simultaneous Multithreading (SMT) processor to speculatively execute multiple paths of execution. When there are fewer threads in an SMT processor than hardware contexts, threaded multi-path execution uses spare contexts to fetch and execute code along the less likely path of hard-to-predict branches.*

This paper describes the hardware mechanisms needed to enable an SMT processor to efficiently spawn speculative threads for threaded multi-path execution. The Mapping Synchronization Bus *is described, which enables the spawning of these multiple paths. Policies are examined for deciding which branches to fork, and for managing competition between primary and alternate path threads for critical resources. Our results show that TME increases the single program performance of an SMT with eight thread contexts by 14%-23% on average, depending on the misprediction penalty, for programs with a high misprediction rate.*

1 Introduction

A primary impediment to high throughput in superscalar processors is the branch problem. Integer programs have on average 4 to 5 instructions between each branch instruction [1]. On today's deeply pipelined processors, the CPU typically has many unresolved branches in the machine at once. This compounds the branch problem, particularly early in the pipeline — the fetch unit will only be fetching useful instructions if all unresolved branches were predicted correctly. Our results show that `li`, despite 94% branch prediction accuracy, is fetching useless instructions 28% of the time, and `go`, with 70% branch prediction accuracy, is fetching useless instructions 63% of the time. The branch problem will become worse as the issue width and the number of pipeline stages to resolve a branch increase for future processors.

Current branch prediction schemes are based on recognizing branching patterns, and are extremely effective for some branches. However, other branches will never fit into this category because they are data-dependent on relatively random data. For those branches, prediction is not a sufficient solution. This paper examines a new speculative execution technique called *Threaded Multi-Path Execution* (TME). TME uses resources already available on a simultaneous multithreading [7, 8] processor to achieve higher instruction level parallelism when there are only one or a few processes running. By following both possible paths of a conditional branch, in the best case we can completely eliminate the misprediction penalty if there is significant progress down the correct path when the branch is resolved.

Simultaneous multithreading (SMT) allows multiple hardware contexts, or threads, to dynamically share the resources of a superscalar processor, even issuing instructions from multiple threads in the same cycle. Multithreading is effective when multiple threads share the system, but does nothing to improve single-thread performance; TME is used for this purpose. Tullsen, et al. [7], showed that SMT can help relieve the branch problem by making the system more tolerant of branch mispredictions, but only in a multiple-thread scenario. Using idle threads on an SMT processor to aid branch prediction is a natural match for the branch problem, because when branch performance is most critical (few threads in execution), the resources we need to attack the problem (idle contexts) are in abundance. The low overhead of contexts on an SMT processor led us to investigate multi-path execution, which differs from dual-path execution by following both paths of multiple branches at once. A key advantage of the SMT architecture is that the physical register file is already shared. Thus, the register state can be "copied" from one context to another by simply copying the register map, which is much smaller than the registers themselves.

This paper examines issues arising from the ability to execute *multiple* alternate paths, multiple predicted paths (i.e., many threads, or processes, active in the system), or both at once. In addition, we detail a potential architecture for threaded multi-path execution on an SMT processor. We use a detailed simulation environment to evaluate results, and explore several alternatives for path spawning and thread priority strategies.

The SMT architecture we use is summarized in Section 2, and modifications to that architecture to support threaded multi-path execution are discussed in Section 3. Section 4 describes the methods we use to evaluate the performance of threaded multi-path execution, and Section 5 presents our results. Section 6 describes related work. We summarize our contributions in Section 7.

2 An SMT Architecture

The TME architecture is an extension to a simultaneous multithreading processor. In this section we describe the baseline SMT architecture, and in the next section the extensions needed to enable threaded multi-path execution.

A simultaneous multithreading processor allows multiple threads of execution to issue instructions to the functional units each cycle. This can provide significantly higher processor utilization than conventional superscalar processors. The ability to combine instructions from multiple threads in the same cycle allows simultaneous multithreading to both hide latencies and more fully utilize the issue width of a wide superscalar processor.

The baseline simultaneous multithreading architecture we use is the SMT processor proposed by Tullsen et al. [7]. This processor has the ability to fetch up to eight instructions from the instruction cache each cycle. Those instructions, after decoding and register renaming, find their way to one of two 32-entry instruction queues. Instructions are dynamically issued to the functional units when their register operands are ready (as indicated by a busy-bit table, one bit per physical register).

The memory hierarchy has 32KB direct-mapped instruction and data caches, a 256 KB 4-way set-associative on-chip L2 cache, and a 2 MB off-chip cache. Cache line sizes are all 64 bytes. The on-chip caches are all 8-way banked. Throughput as well as latency constraints are carefully modeled at all levels of the memory hierarchy. Conflict-free miss penalties are 6 cycles to the L2 cache, another 12 cycles to the L3 cache, and another 62 cycles to memory.

Branch prediction is provided by a decoupled branch target buffer (BTB) and pattern history table (PHT) scheme. We use a 256-entry, four-way set associative BTB. The 2K x 2-bit PHT is accessed by the XOR of the lower bits of the address and the global history register [5, 13]. Return destinations are predicted with a 12-entry return stack (per context).

The instruction fetch mechanism we assume is the ICOUNT.2.8 mechanism from [7]. It fetches up to eight instructions from up to two threads. As many instructions as possible are fetched from the first thread; the second thread is then allowed to use any remaining slots from the 8-instruction fetch bandwidth. The ICOUNT fetch priority mechanism gives priority to those threads that have the fewest instructions in the processor between the fetch stage and instruction issue.

We assume a 9-stage instruction pipeline, which is based on the Alpha 21264 pipeline, but includes extra cycles for accessing a large register file, as in [7]. This results in a misprediction penalty of 7 cycles plus the number of cycles the branch instruction stalled in the processor pipeline. Instruction latencies are also based on the Alpha.

2.1 Register Mapping Architecture

For register renaming, the SMT architecture uses a mapping scheme (derived from the MIPS R10000 [12]) extended for simultaneous multithreading as shown in Figure 1 (except for the mapping synchronization bus, which is added for TME and described in Section 3.2). There are three parts to the mapping architecture (1) a mapping table, which contains the current virtual-to-physical register mappings, (2) active lists, which maintains the order of active instructions and holds register mappings that may become invalid when the instruction completes, and (3) a list of free registers.

Each context in an SMT processor requires its own *mapping region* to map the virtual to physical registers. Mapping tables in existing processors are shadowed by *checkpoints*, which are snapshots of the mapping region taken when a branch was encountered. A branch misprediction causes the register mapping to be restored from a checkpoint. With this scheme, each thread can independently save and restore checkpoints without contention. In our baseline implementation, each context's mapping region has eight checkpoints, and the thread stalls when it runs out of checkpoints.

Each context has an active list so that instructions can commit independent of the progress of other threads. Instructions are entered into this list in-order during decoding and are squashed or committed in-order as branches are resolved. Each instruction that writes a virtual register removes a physical register from the free list and writes a new mapping into the mapping table. A pointer to this physical register is stored with the instruction in the active list along with a pointer to the old physical register, which represents the previous virtual to physical register mapping. An instruction that commits from the end of an active list frees the old physical register used by the previous mapping by returning it to the free list. A squashed instruction frees the physical register from the new mapping.

3 Threaded Multi-Path Execution

Threaded multi-path execution uses unused (spare) contexts to execute threads along alternate paths of conditional branches. As a result, the SMT's resources can be more fully utilized, and the probability of executing the correct path is increased.

An effective implementation of TME must be able to (1) accurately identify good candidates for spawning, (2) be able to start the alternate-path thread running in a separate hardware context on the SMT processor, and (3) provide efficient

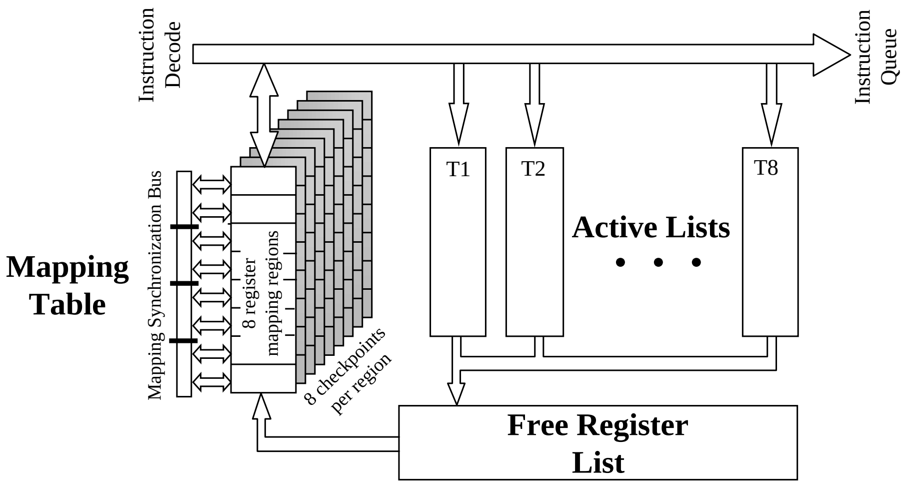


Figure 1: A register mapping scheme for a simultaneous multithreading processor, with an added mapping synchronization bus to enable threaded multi-path execution.

instruction fetching of both primary and alternate paths so that alternate-path threads do not degrade the performance of the processor. A more detailed description of path selection and the instruction fetch policies are provided in Sections 5.2 and 5.3. This section concentrates on describing the functionality of the thread spawning architecture.

The following terms are used throughout the rest of the paper to describe the behavior of the TME architecture: *Primary-path thread* — the thread that has taken the predicted path through the code; *alternate-path thread* — a thread that is executing an unpredicted path through the code; *idle context* — a context that is not executing any thread; *spare context* — a context that is partitioned to a primary thread to be used for alternate path execution; *fork a branch* — take both paths of a branch; *spawn a thread* — create an alternate-path thread; *fork a checkpoint* — use the checkpoint of a branch to start running the alternate-path thread on a spare context.

3.1 Path Selection

We use two mechanisms to pare down the number of potential branches to fork. First, alternate paths are forked only along the primary path. This prevents forking from an alternate path, which simplifies the architecture. Branches along alternate paths are predicted in the normal fashion.

Second, to determine which primary-path branches to fork, we added branch confidence prediction (as described in [3]) to the SMT architecture. A 2048 entry table of n-bit counters, shared among all hardware contexts, is used to keep track of the predictability of a branch. The confidence table is indexed by XORing the PC with the global history register, the same index used for the branch prediction table. The n-bit counter is a saturating up counter and is incremented each time a branch is correctly predicted. If the branch is incorrectly predicted, the counter is set back to zero; the counter is essentially a count of the correct predictions since the last mispredict for this entry. When the counter has a value less than some threshold, it is identified as having low confidence. In Section 5.2 we examine the effects of branch confidence on TME by varying this threshold.

3.2 Mapping Synchronization Bus

An alternate path can be spawned if there exists an idle context. When an alternate path is spawned, the idle context it uses needs to operate on mapping information that reflects the state just after the forking branch (which is either the current mapping region or a checkpoint). The *Mapping Synchronization Bus* (MSB) copies the current mapping region between thread contexts. This is a bus which is connected to all the contexts' mapping regions, as shown in Figure 1. If there are between 256 and 512 physical registers, a map copy requires the movement of 288 bits (32 entries by 9 bits) for 32 logical registers. A 36-bit bus could accomplish that in 8 cycles, a wider bus in fewer cycles (The effect of copy latency is discussed in Section 5.1). This requires additional read/write ports for the mapping table. Information is read from the source region, broadcast over the bus, and written to destination regions.

The MSB synchronizes an idle context's register map with the primary thread's. To accomplish this, the MSB constantly broadcasts the primary thread's register mapping region (looping around the region) to its spare contexts. Once the mapping region of an idle context is synchronized with the primary mapping table, it remains synchronized by snooping updates to the primary context's mapping table. It will then be ready to spawn an alternate path immediately when an appropriate branch is encountered. If the alternate path is needed before the synchronization completes, alternate-path fetching can still begin two cycles before the copy completes because the rename stage is two cycles after the fetch stage. When a branch is forked, neither the primary or alternate context need to checkpoint the branch, as the alternate thread itself acts as the checkpoint.

To allow multiple primary paths to have alternate threads at the same time, the thread contexts must be partitioned, giving each primary thread a set of spare hardware contexts to use as alternate path threads. Because of the broadcast nature of copying mapping tables across the MSB, the MSB must also be partitioned in hardware. Transmission gates placed on the MSB between the thread hardware contexts can dynamically be turned on allowing flow across the bus, or turned off creating independent local buses on each side of the gate.

We assume an architecture which can dynamically partition the MSB between 1 to 4 primary threads. Our MSB architecture has 3 transmission gates located where the lines cross the MSB in Figure 1. This allows us to create two partitions (e.g., contexts 1-4,5-8, by turning the middle transmission gate off), three partitions (e.g., 1-2, 3-4, 5-8) or four partitions (1-2, 3-4, 5-6, 7-8). We expect that the partitioning will change at very coarse intervals (e.g., when a new process comes into the system), and threads can move quickly between contexts using the same spawning mechanism described, so the current location of a thread will never inhibit our ability to use TME for a new thread or to equitably repartition the processor. A thread not using TME ignores the MSB, so we can still have 8 independent threads in the system.

3.3 Spawning Alternate Paths

When there is an idle hardware context available for use, an alternate thread can potentially be spawned. The TME policies for using an idle context to spawn alternate path threads can be summarized in four cases:

1. If there are no candidate checkpoints to be forked, the idle context's mapping region is synchronized with the primary path's using the MSB. The next branch selected to fork would have its non-predicted path spawned as the alternate path for this idle context. This is the common case when there are an abundance of idle threads available.

2. If there exists a candidate checkpoint waiting to be forked and the checkpoint is on the hardware context of the primary path, the idle context's mapping region synchronizes with the primary path's using the MSB. The synchronized idle context then becomes the primary path. This frees up the hardware context that contains the checkpoint, so that context can start executing the checkpoint as an alternate path.

3. If a candidate checkpoint is waiting to be forked and it is stored on a newly idle context, then the idle context starts executing the checkpoint as an alternate path. This can occur when in case (2) above the primary path migrates from one thread context to another leaving fork-able checkpoints behind.

4. If the primary path runs out of checkpoints, the primary path will switch to an idle context after it is synchronized, allowing the primary path to use more checkpoints.

This scheme never requires that we copy checkpoints, only the current mapping region of the primary thread, which greatly simplifies the hardware.

Figure 2 shows a detailed example involving the first three scenarios. In this Figure, there are three hardware contexts numbered one to three from left to right, and one primary path. Each hardware context has two checkpoints, shown in the shaded boxes. Each Figure shows the paths and branches executing, along with the current mapping table with checkpoints. `PT` stands for primary thread; `AT` stands for alternate thread; `IT` stands for idle thread. In the flow graphs, the thick lines represent the primary path thread, the thin lines represent alternate path threads, and the dotted lines represent un-spawned paths.

To start, Figure 2(a) shows a primary path `A` executing on the first hardware context, and the two idle contexts have been synchronized with `A`'s mapping table via the MSB. Figure 2(b), shows the execution state after encountering branch `b1`, and TME elects to spawn the other path of branch `b1`. Since the second hardware context is synchronized with the primary path, the alternate path `B` starts executing on this context. The branch `b1` does not need to be checkpointed, because both of its paths are executing.

Figure 2(c) shows the execution state after a branch for both path `A` and path `B` have been encountered. Assuming TME predicts that `b2` should be forked, the alternate path `C` will start executing on the third hardware context. Branch `b3` is not an option for forking because it is on an alternate path.

Figure 2(d) shows the execution state after all three paths have encountered another branch. The primary path `A` cannot fork branch `b4` even if it is a candidate because there are no idle thread contexts; therefore, the branch is checkpointed.

Now we assume that branch `b1` is resolved and it was mispredicted. Therefore, path `B` was the correct path of execution. Figure 2(e) shows the resulting mapping table and flow graph after the instructions and branches from paths `A` and `C` are squashed. The first and third hardware contexts are now idle, and TME synchronizes their mapping regions with the primary path `B` using the MSB.

After the first and third hardware contexts are synchronized, they can start running an alternate path. Assume that TME predicts that both branches `b3` and `b6` should be forked, with `b3` having the highest priority. Since there is an idle context and the checkpoint to be forked exists on the same context as the primary path, the primary path of exe-

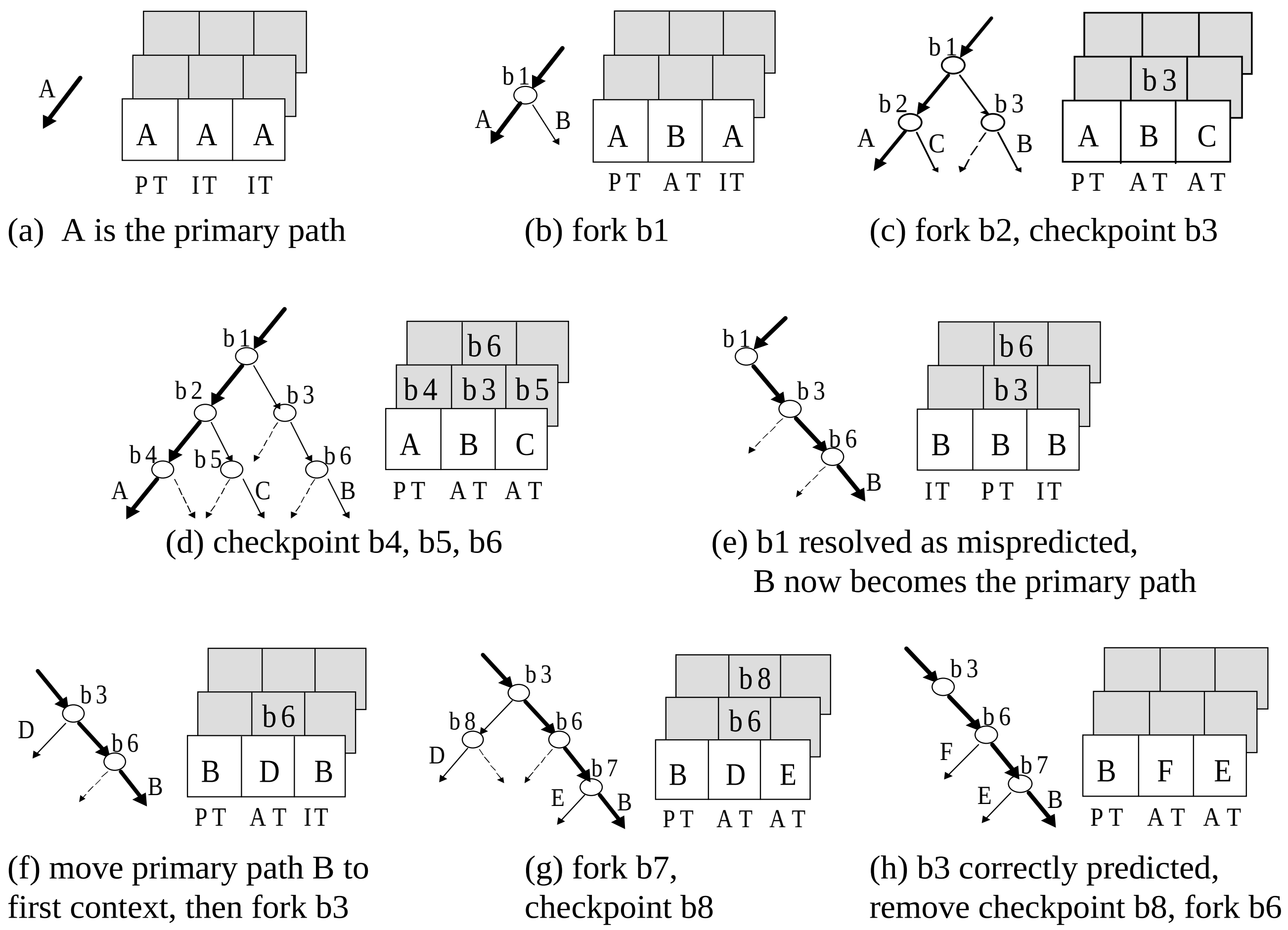

(a) A is the primary path
A
PT IT IT
(b) fork b1
b1
A
B
PT AT IT
(c) fork b2, checkpoint b3
b1
b2
b3
C
PT AT AT
(d) checkpoint b4, b5, b6
b4
b5
b6
PT AT AT
(e) b1 resolved as mispredicted, B now becomes the primary path
IT PT IT
(f) move primary path B to first context, then fork b3
D
PT AT IT
(g) fork b7, checkpoint b8
b7
b8
E
PT AT AT
(h) b3 correctly predicted, remove checkpoint b8, fork b6
F
PT AT AT

Figure 2: Example of spawning alternate paths and using the mapping synchronization bus. `PT` stands for primary thread; `AT` stands for alternate thread; `IT` stands for idle thread. In the flow graphs, the thick lines represent the primary path thread, the thin lines represent alternate path threads, and the dotted lines represent un-spawned paths.

cution `B` is moved from hardware context two to one. This frees the second context where the checkpoint resides to fork checkpoint `b3`, executing path `D` (Figure 2(f)). It must not overwrite checkpoint `b6` as it executes, however. Checkpoint `b6` cannot be forked yet in this example, which is a constraint resulting from the fact that we never copy checkpoints.

Figure 2(g) shows the execution state after paths `B` and `D` have each encountered another branch. If the primary path `B` elects to fork branch `b7`, path `E` starts running on idle context three. Finally, assume branch `b3` is resolved, and was correctly predicted. The checkpoint `b8` is removed from the second hardware context, but the checkpoint for branch `b6` remains since it is from the primary path. Since branch `b3` was correctly predicted, the second context becomes idle, but it contains the checkpoint `b6`. If the branch `b6` is not selected to fork, then the idle hardware context would resynchronize with the primary path `B` using the MSB. But in this case, TME predicts branch `b6` should be forked. Then the path `F` is spawned in the second hardware context, and the checkpoint `b6` is removed (Figure 2(h)).

As shown in this example, a contribution of our spawning algorithm is that it reduces the utilization of checkpoint resources by (1) not storing a checkpoint when a branch is forked, (2) distributing the checkpoints among primary path and alternate path threads, and (3) allowing a primary path thread that has run out of checkpoints to switch to an idle context.

3.4 Issues

A hardware context becomes idle when a forked branch is resolved. To resolve branches in TME, each executing branch must store with it the location of its "checkpoint," which may be a physical checkpoint or a context running on an alternate path. When the branch is resolved with a correct prediction, either the checkpoint is deleted or the spawned path is squashed. When it is resolved with an incorrect prediction, either the context containing the checkpoint becomes the primary path starting at that checkpoint, or the thread running the alternate path of the branch becomes the primary thread.

When a branch forks, a physical register can be used in both paths, since both paths start with identical mapping tables. Also, physical registers defined before the forked branch may be marked for freeing by instructions down both paths. However, this requires no changes to our architecture, because only one of the paths will be the correct path and only the instructions on the correct path will commit. Therefore, the register that the committing instruction frees is guaranteed to be dead along every existing path.

Speculative memory dependencies also become more complicated. A load down an alternate path could be dependent upon a speculative store executed on the primary path prior to spawning the alternate path thread. Determining which stores a load may or may not depend on is complex. We have considered two alternative techniques for handling speculative stores, which are briefly described in the following two paragraphs. For a complete description see [11].

The first approach assumes a unified non-speculative store buffer, and separate per-thread speculative store buffers (this division could be physical or virtual). A load on an alternate path would have to search both its own and (part of) its primary path's store buffers for possible dependencies. This scheme could limit our ability to spawn new threads shortly after resolution of a mispredicted branch if it is not the oldest unresolved branch. This is because the primary thread could occupy multiple speculative store buffers until the older branches are resolved. This effect should be small.

The second approach assumes a single unified store buffer. To identify which hardware context or path executed a speculative store, we associate with each store in the buffer a speculative tag of N bits. Each bit represents a unique Id identifying the hardware context or path that executed the speculative store instruction. The Id bits are similar to the idea of context tags in [4], but our implementation and use is different since TME can have multiple processes simultaneously executing. For TME, each hardware context has a *Speculative Tag Id* (STI) and a *Search Mask* (SM) to handle speculative stores. The STI represents a *single* asserted Id bit, which uniquely identifies the hardware context executing the store and the prior branch of the store. The SM is used to accumulate search information up to the current point of execution for the thread running on the hardware context. Each path following an executed branch has its hardware context STI set to a new unused Id bit. The SM for the hardware context is then set to be the SM of the prior path ORed with the new STI bit. When a store is executed, the stores speculative tag is set with the hardware context's STI. When a load is executed, the load's SM is ORed with each tag for all of the speculative store instructions to identify stores that must be checked for aliasing. When a path (branch) is resolved, its STI bit is reclaimed by broadcasting it to each hardware context and clearing the bit in each context's SM. In addition, a search of the store buffer is performed and if a store's speculative tag matches the STI bit the store is either marked as non-speculative or squashed based on the outcome of the branch.

4 Evaluation Methods

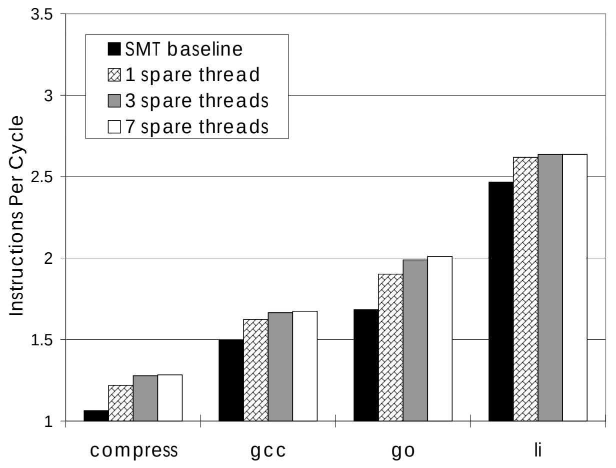
Our measurement methodology allows us to model in detail the base architecture described in the previous section, with the many extensions described in the next section.

Our simulator is derived from the betaSMT simulator [6]. It uses emulation-based, instruction-level simulation. The simulator executes unmodified Alpha object code and models the execution pipelines, memory hierarchy, TLBs, and the branch prediction logic of the processor described in Section 2. This simulator accurately models execution following a branch misprediction. It fetches down incorrect paths, introduces those instructions into the instruction queues, tracks their dependences, and possibly issues and executes them. We extended this capability to allow the spawning of speculative threads on path forks. Both the primary-path and alternate-path threads have the ability to mispredict branches further down the instruction stream and later recover. Our throughput results only count useful (committed) instructions.

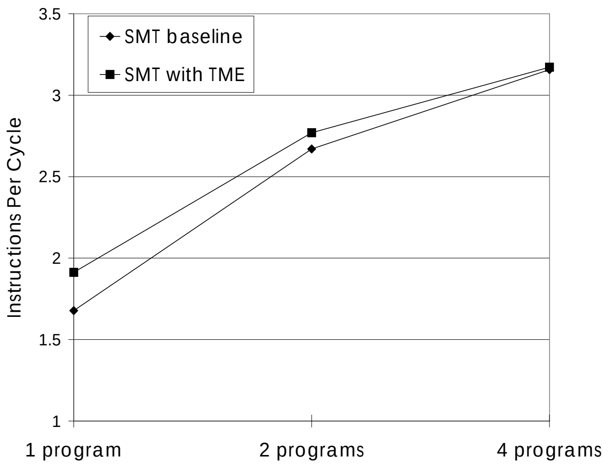
Our workload primarily consists of `compress`, `gcc`, `go` and `li` from the SPEC95 benchmark suite. We chose these four programs because they show varying levels of IPC (1 to 2.5) and they have high branch misprediction rates of 20%, 18%, 30%, and 6% respectively for the length of our runs. Our goal for this architecture is to achieve significant improvement on workloads that suffer from poor branch behavior, but without degrading the performance of workloads that do not. We demonstrate the former by focusing most of the paper on a workload biased toward low branch prediction accuracy. We demonstrate the latter in Section 5.5, where we examine applications with few mispredictions, as well as mixed workloads. We compiled each program with the Digital Unix C compiler under DEC OSF V4.0, with full optimization (-O5 -ifo -om). Each program is simulated from its start for 100 million committed instructions.

5 Results

This section presents the performance of TME under a number of different scenarios and architectural alternatives. It examines (1) the effects of different mapping synchronization latencies, (2) hardware mechanisms for selecting which alternate paths to spawn, (3) the instruction fetch priorities used for TME, (4) the effect of branch misprediction penalties, (5) TME performance for programs with differ-



(a) Performance from TME using 1 primary thread with 1, 3, or 7 spare contexts.



(b) Average performance of TME in the presence of 1, 2 and 4 primary threads.

Figure 3: Performance improvement from TME.

ent branch misprediction characteristics, and (6) the effect of allocating the spare contexts dynamically.

Figure 3 shows the improvement over the original SMT architecture (*SMT baseline*) of the default TME architecture configuration. These programs average a 14% gain over the baseline. The TME default configuration shown in this Figure assumes a latency of 4 cycles for resynchronizing the register map, a minimum branch misprediction penalty of 7 cycles, a fetch priority for alternate threads of $8 + 4 * confidence\_counter$, the primary-variable heuristic with a *confidence_multiplier* of 4 for choosing which branch to fork when more than one candidate exists, and a branch confidence counter threshold of 8. All of these will be explained later in this section. The default configuration is composed of our top choice for each of the options we will examine.

Figure 3(a) shows the performance for the four programs `compress`, `gcc`, `go`, and `li`, when one primary path is running with 1, 3 and 7 spare contexts. The performance of 3 spare contexts is nearly the same as 7. Always having a spare context ready to fork provides the best performance on our architecture because it hides the synchronization latency. Three spare contexts is usually enough to allow this, but one spare context is not.

Figure 3(b) shows the performance of TME with 1, 2 and 4 programs running on the processor at the same time. Results are shown for the average of the four programs in Figure 3(a). The performance for TME with 1 primary thread uses 7 spare hardware contexts, 2 primary threads have 3 spare contexts each, and 4 primary threads have 1 spare context each. TME does not degrade the performance of multiple programs running on an SMT processor, but TME is less effective when there are more programs running, as expected, because (1) there are fewer alternate contexts available, (2) the branch problem is less severe when the workload is multithreaded, and (3) there are fewer resources available to the alternate-path threads.

In the rest of this section, single program results use all eight hardware contexts (one primary and seven spare) because that matches our assumed SMT architecture. The multiple program results in this section correspond to the average of all permutations (of a particular size) of the four programs `compress`, `gcc`, `go`, and `li` evenly partitioning the spare contexts among the multiple programs. For all the results, the TME approach used is always the default configuration described above, except for the mechanism being varied.

5.1 Mapping Synchronization Latency

It takes time to duplicate register state for multiple path execution, and this latency can have a significant impact on the viability of multiple-path or dual-path execution. Because the SMT architecture shares physical registers, we can copy register state simply by copying the register map. Figure 4 shows the effects of latencies of 0, 4, 8 and 16 for resynchronizing the register mapping across the MSB for the mapping table synchronization architecture described in Section 3.2. This is the latency across the bus. The alternate thread can typically begin fetching two cycles before the map is synchronized, since the renaming stage is two stages after fetch; therefore, we can also get the *latency 0* performance with a MSB latency of 2 cycles. With a latency of 8 cycles or worse, the performance begins to degrade, especially in the case of `go`. Nevertheless, the results show that the MSB is an effective method to hide latency.

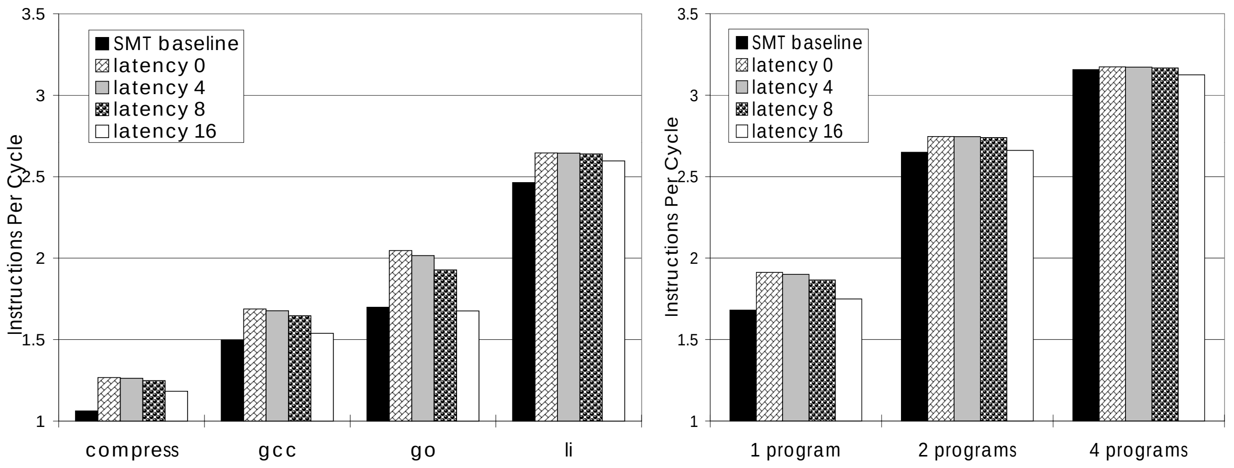


Figure 4: Effects of mapping synchronization with a latency of 0, 4, 8 and 16 cycles.

5.2 Thread-spawning Strategies

Not all branches are equally good candidates for forking, so we would like to choose the best candidates given our limited resources. Branch selection involves two different decisions: (1) identifying which branches are good candidates for forking and (2) selecting among these when there are multiple candidates.

5.2.1 Identifying Candidate Branches

The confidence prediction mechanism identifies each branch as either a low confidence or high confidence branch by examining the confidence counter described in Section 3.1.

If the counter is equal to or below a *fork-threshold*, then the branch path is a candidate for being spawned. Otherwise, the branch is not considered for forking. Figure 5 shows the performance with a fork-threshold value of 0, 4, 8, and 12. To gather these results, the size of the n-bit branch confidence predictor was equal to $\lceil lg(fork\ threshold\ value) \rceil$ bits. Lower thresholds make the system more selective. Also, the figure shows the performance when branches are always considered for forking (*always*) and when only mispredicted branches are forked (*oracle*). A counter value of 8 or 12 yielded the best non-oracle performance. Always forking branches gave strong performance as well, without the added cost of a confidence predictor table. These results indicate that with a single program and eight contexts, an SMT processor has enough spare resources that a fairly liberal spawning strategy works best. About half the potential increase from original to oracle performance has been achieved. These results imply that improving the branch confidence techniques used to identify candidate branches is an important area of future research.

5.2.2 Candidate Selection

When more candidates for forking exist than spare contexts are available, the best candidate should be chosen. This happens when there are frequent candidate branches or few spare threads — a context becomes free and observes multiple candidate branches desiring to be forked. We examined 4 alternatives for selecting a branch to fork:

- oldest — fork the oldest unresolved branch that has been identified as a candidate.
- latest — fork the most recent unresolved branch that has been identified as a candidate.
- next — when a thread becomes idle, fork the next low-confidence branch encountered during execution ignoring previous checkpoints.
- lowest-confidence – fork the branch with the lowest confidence fork-threshold among those identified as candidates.

Figure 6 shows the performance for the fork heuristics. The Next heuristic performed poorly. The Oldest, Latest, and Lowest-Confidence heuristics all performed about the same for the 1-program results because (1) plenty of spare hardware contexts are typically available with this configuration and (2) our register mapping synchronization scheme allows us to postpone the spawn decisions until after the mapping is resynchronized (reducing the cost of a bad decision). For multiple programs, the *oldest* and *lowest* schemes have a slight edge over the others.

5.3 Fetch Priority

All execution resources are shared on an SMT processor, but with TME we do not necessarily want to allocate resources evenly to all threads. Typically, the primary-path threads still

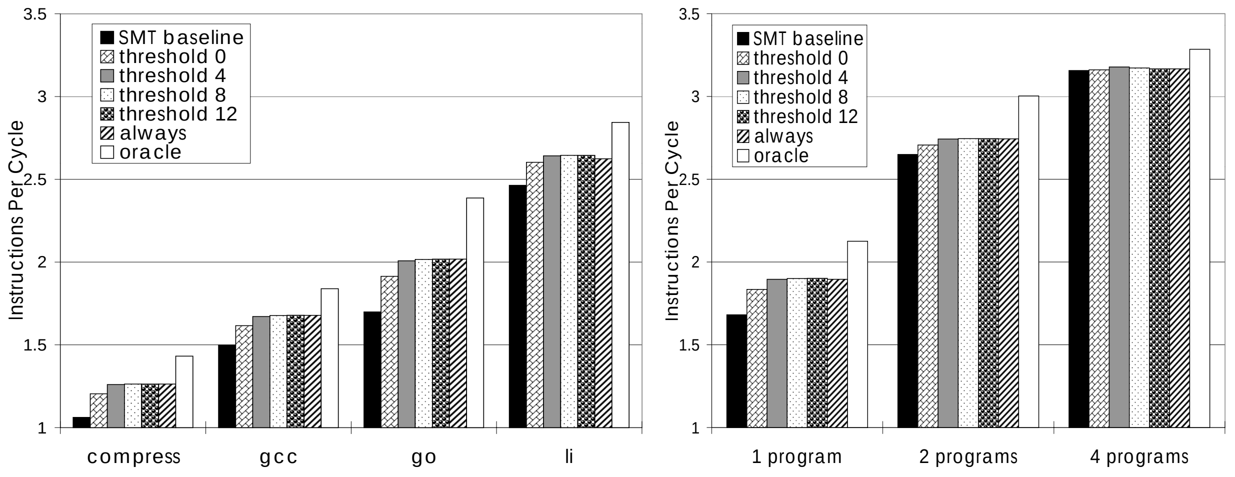


Figure 5: Performance of candidate identification

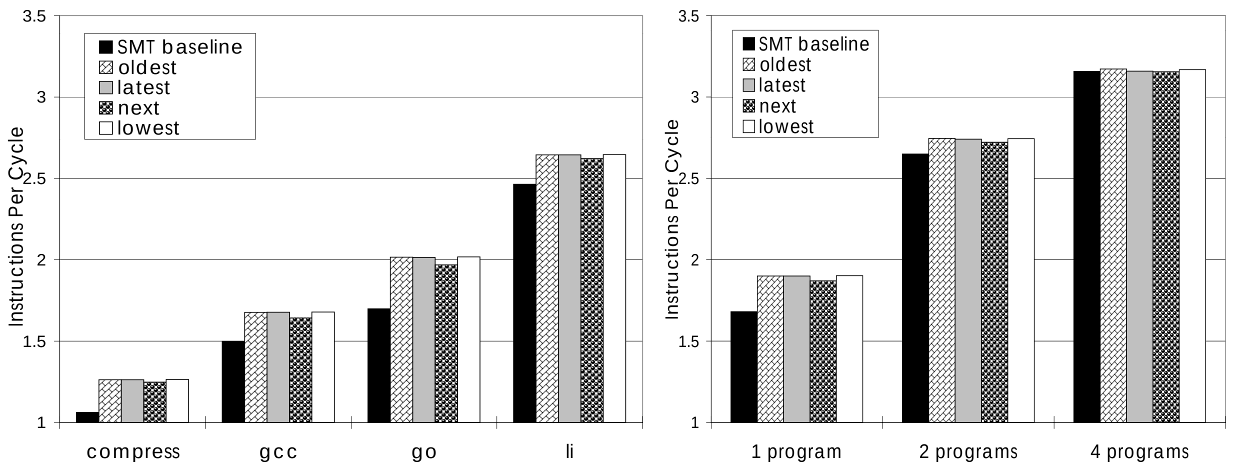


Figure 6: Performance of candidate selection

represent the most likely paths, and the impact of alternate-path threads should be kept low.

The fetch mechanism is the most logical place to control the allocation of resources in the processor for two reasons. First, the fetch bandwidth has often been identified as the bottleneck resource both in single-thread and multiple-thread processors. Second, the fetch unit is the gateway to the processor — giving a thread priority access to the fetch unit gives it priority access to the entire machine. Tullsen et al. [7] showed that assigning fetch priorities for threads led to significant performance gains. With TME, the variances between the “usefulness” of threads are even greater.

In our baseline simultaneous multithreading processor, the ICOUNT [7] fetch scheme gives highest priority to threads with the fewest un-issued instructions in the machine. It uses a counter associated with each thread to determine which two threads get to fetch, and which of those have higher priority. Because the scheme is based on counters, it is easy to artificially adjust a thread’s priority by bumping its counter up or down.

The key feature of this mechanism is that it assigns priorities dynamically every cycle. Bumping a context’s counter up by n means that that thread will only fetch when other threads have n more instructions in the machine than this one. This, for example, allows us to only fetch from an alternate-path thread when primary-path threads have accumulated enough ILP (i.e., instructions in the processor) to run for a few cycles without missing the lost fetch bandwidth.

We examine the following policies for setting fetch priorities. Note that adding a value to a thread’s counter *lowers* its priority.

- high-constant — add a constant to the priority fetch counter for the alternate-path threads that is higher than the maximum possible value for primary-path threads. This partitions the threads into two distinct groups. Alternate-path threads will only fetch when no primary-path threads are available (e.g., because of

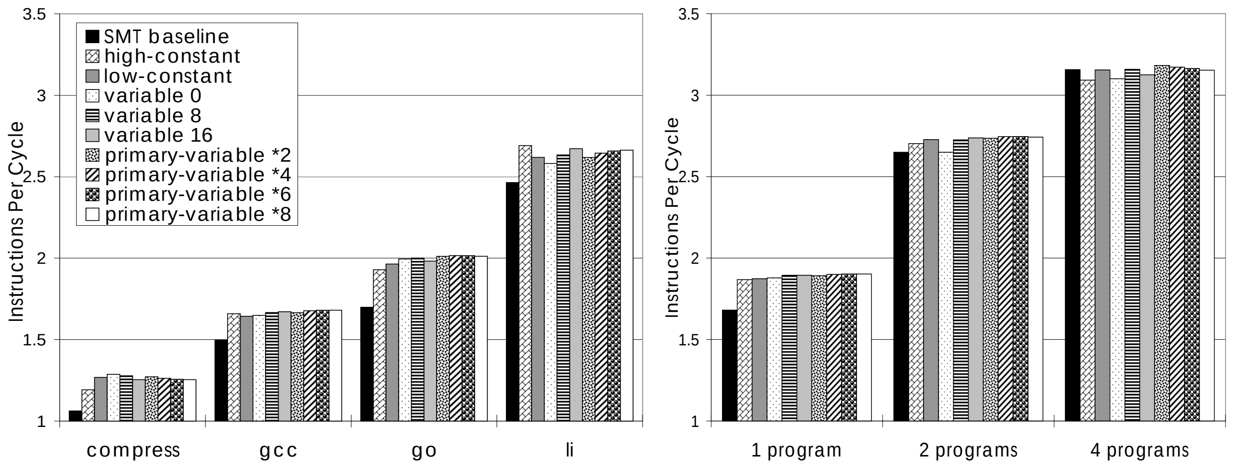


Figure 7: Performance of TME fetch priority policies

an instruction cache miss). This minimizes the impact of alternate-path threads on primary-path throughput. This scheme initializes the fetch priority counter to the constant value when an alternate-path is spawned, and subtracts that value if an alternate-path becomes the primary path. The high-constant value used is 64 (the combined size of the instruction queues).

- low-constant — similar to high-constant, but this scheme adds a lower value (16). If the primary-path thread has greater than 16 more instructions in the machine than the alternate-path, the alternate-path will temporarily have higher fetch priority than the primary thread.

- variable — this scheme adds an amount to the priority counter that is an indication of branch confidence for the forked branch. For a low-confidence branch, a low value is added to the counter for the newly forked alternate-path thread; high-confidence branches add a higher value. For the results shown, a constant of 0, 8, or 16 is used in the equation *constant + 4 * confidence_counter* for the initialization of the fetch priority. For these results, a 3-bit confidence counter is used.

- primary-variable – this scheme includes the current priority count of the primary-path into the initialization of the alternate-path's fetch priority. The alternate-path's fetch priority is initialized to *primary_path_priority_count* + *confidence_multiplier* $*$ *confidence_counter*. We examine confidence multiplier values of 2, 4, 6, and 8.

Figure 7 shows the performance of these four fetch priority schemes. We observe that the performance behavior is different for each program, and there is no clear winner. The constant chosen has a significant effect on performance for individual programs. For instance, `gcc` and `li` (with higher prediction accuracy in general) perform better with a high constant (a conservative policy). In contrast, `compress` and `go` perform better with a low constant (an aggressive approach). This implies that the variable schemes should perform better and more consistently, which is shown to some extent in the results. Not only do the variable schemes allow better primary/alternate priority management, but they also allow the system to differentiate the priority of multiple alternate-path threads. The primary-variable policy uses the most run-time information. As a result, it has good performance for all programs tested. Primary-variable 4 was chosen for inclusion in our default TME configuration.

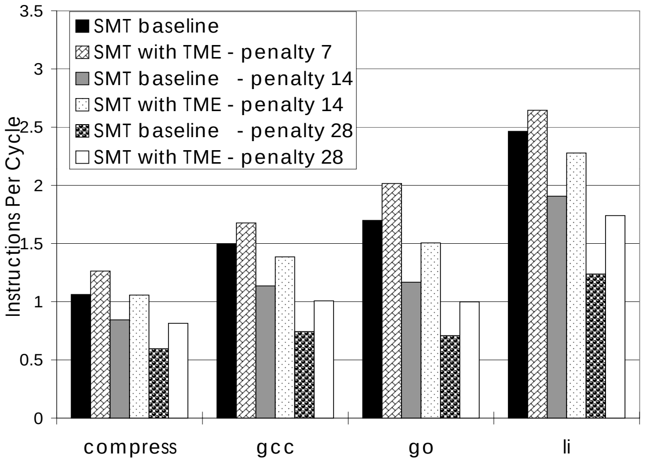
5.4 Branch Misprediction Penalties

Many existing processors have higher branch misprediction penalties than the pipeline we have been assuming, and it is likely that branch penalties will continue to increase. In this section, we examine the effect of increasing the minimum branch misprediction penalty from 7 cycles to 14 and 28 cycles. The penalty for a mispredicted branch is this minimum penalty plus the number of cycles the branch stalls waiting in the issue queue.

In Figure 8, the performance improvement of TME increases from an average of 14% to over 23% for single program performance when increasing the misprediction penalty to 14 cycles. It should be noted that the importance of TME to multiple-primary-thread performance increases significantly with a higher misprediction penalty.

5.5 Varying Branch Prediction Rates

Our workload is intentionally heavy on programs with poor branch performance, but we are also concerned that we do not degrade the performance of programs that already have good branch accuracy. Figure 9 (the static allocation results)



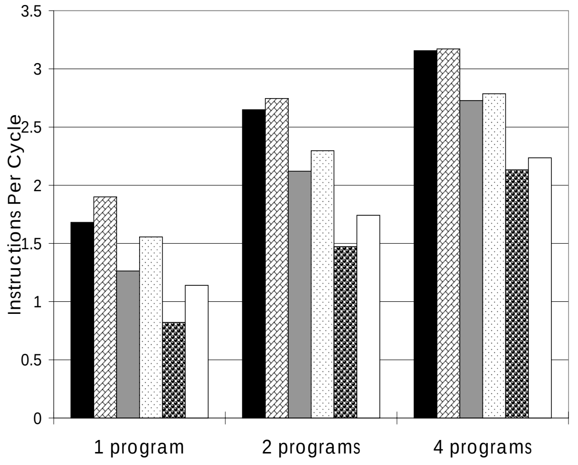


Figure 8: Effect of a high misprediction penalty

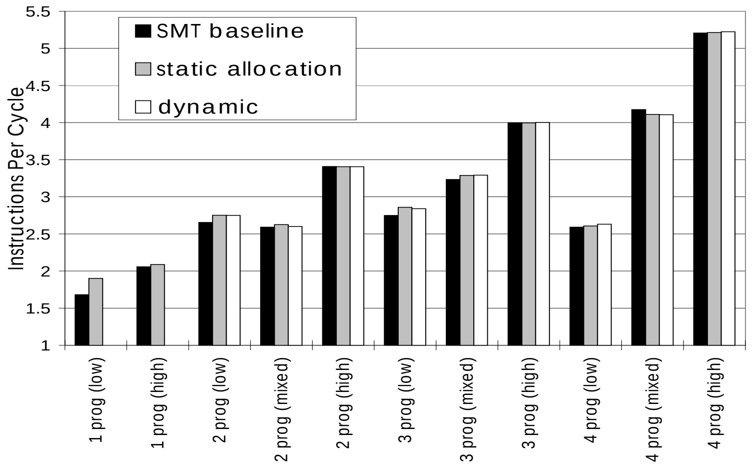


Figure 9: Performance for different branch prediction accuracies and dynamic allocation of spare contexts.

shows the performance of TME for three groups of programs with different branch prediction accuracies. The programs included in the *low* group are `compress`, `gcc`, `go` and `li`. The programs in the *mixed* group include `go`, `compress`, `fpppp`, and `tomcatv`. The programs in the *high* group include `fpppp`, `tomcatv`, `alvinn` and `ora`. For the new programs, the branches are predicted with an accuracy of 94% for `fpppp`, 99% for `tomcatv`, 96% for `alvinn`, and 99% for `ora`. Average results are shown for running all combinations for each group of programs. The results show that the performance improvement of TME decreases for programs with a high branch prediction accuracy. In only one case is there a slight overall degradation with TME; in general it consistently either helps or does not inhibit performance, depending on the needs of the workload. The reason is that the mechanisms for deciding when to spawn alternate paths uses branch confidence prediction, and it chooses to spawn fewer paths for the “high” group of programs.

5.6 Dynamic Allocation of Spare Contexts

When more than one primary thread is executed simultaneously, the baseline architecture evenly partitions the spare contexts as described in Section 3.2. Even partitioning may not yield the best performance. We may want to devote more idle threads to programs with poor prediction accuracy, stealing from those with high accuracy.

We examined a scheme which does dynamic allocation of spare contexts, instead of evenly partitioning the spare contexts. While our architecture has some ability to do uneven partitioning (e.g., you could give 2 programs no spares, one a single spare, and one three spares), we have chosen in this section to consider any possible partition to see if there is a performance benefit from dynamic allocation. Note that this level of flexibility would greatly complicate the MSB architecture since the simple transmission gate partitioning approach would no longer work.

Every 10,000 cycles, the dynamic allocation evaluates the behavior of the programs and adjusts the allocation of spare contexts. The basis for its decision is the number of mispredicted branches which are within the confidence threshold. This does not necessarily mean these branches were actually forked, but if the branch was a candidate and was later found to be mispredicted, it is included in this count. The evaluation process considers the total number of spare contexts among all primary threads and then allocates the spares to primary threads proportional to the percentage of candidate mispredicted branches. Figure 9 shows that for the mix of programs we examined, and this repartitioning algorithm, that dynamic allocation did not perform better than the default static partitioning.

6 Related Work

This research was motivated by the speculative execution technique called Disjoint Eager Execution (DEE) proposed by Uht et al. [10]. Instead of speculatively predicting a single-path, DEE in hardware contains an elaborate structure which allows the processor to speculatively execute down multiple paths in a program. The model for the DEE architecture is different than simultaneous multithreading, in that each of the speculative paths being executed on the DEE architecture has its own processing unit. The DEE architecture adds to a traditional superscalar processor a significant amount of extra state registers needed for each speculative path, but multiple threads cannot take advantage of this extra state simultaneously as in SMT. As the results in this paper have shown, the SMT architecture allows for an efficient extension for threaded multi-path execution.

Our work also draws from the two studies by Heil and Smith [2] and Tyson, Lick and Farrens [9] on restricted dual path execution. Identifying candidate branches for forking through different branch confidence prediction schemes was examined in great detail in both of these studies. In addition Heil and Smith proposed the oldest, latest and next candidate selection techniques, which we examined for selecting which branches to fork for TME.

7 Summary

This paper presents Threaded Multi-Path Execution as a mechanism to reduce branch penalties for single-thread performance by executing multiple paths in parallel. TME provides a cost-effective approach for increasing single thread performance for threaded architectures like simultaneous multithreading by using their otherwise idle resources.

In this paper we describe a new architecture feature, the Mapping Synchronization Bus, which allows TME to efficiently copy register maps to spawn multiple paths. We examine new approaches for selecting which branches to fork, predicting when to spawn alternate paths, and different fetch priorities to partition hardware resources among primary and alternate path threads.

Our results show that TME achieves 14% single-program performance improvement on average using a branch misprediction penalty of 7 cycles for programs with a high misprediction rate. With two programs, we see a 4% improvement. Greater improvements are seen as the misprediction penalty increases. We also show that on an SMT/TME processor running a single program, liberal spawning strategies are most effective, and that dynamic fetch priority schemes that use run-time information, particularly branch confidence measures, increase the effectiveness of TME.

Acknowledgments

We would like to thank Dirk Grunwald, Jack Lo, Susan Eggers, and the anonymous reviewers for providing useful comments. This work was funded in part by NSF CAREER grant No. MIP-9701708, UC MICRO grant No. 97-018, and a DEC external research grant No. US-0040-97.

References

[1] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4), 1994.

[2] T.H. Heil and J.E. Smith. Selective dual path execution. Technical report, University of Wisconsin - Madison, November 1996. `http://www.ece.wisc.edu/~jes/papers/sdpe.ps`.

[3] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *29th Annual International Symposium on Microarchitecture*, pages 142–152, December 1996.

[4] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[5] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-WRL, June 1993.

[6] D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[7] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23nd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[8] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[9] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Technical Report CSE-TR 346-97, University of Michigan, 1997.

[10] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *28th Annual International Symposium on Microarchitecture*, pages 313–325. IEEE, December 1995.

[11] S. Wallace, B. Calder, and D.M. Tullsen. Threaded multiple path execution. Technical Report CS97-551, University of California, San Diego, 1997.

[12] K.C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.

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