Microarchitectural Support for Precomputation Microthreads

Robert S. Chappell† Francis Tseng‡ Adi Yoaz# Yale N. Patt‡

†EECS Department
The University of Michigan
Ann Arbor, Michigan 48109-2122
robc@eecs.umich.edu

‡ECE Department
The University of Texas at Austin
Austin, Texas 78712-1084
{tsengf, patt}@ece.utexas.edu

Texas Development Center
Intel Corporation
Austin, TX 78746
adi.yoaz@intel.com

Abstract

Research has shown that precomputation microthreads can be useful for improving branch prediction and prefetching. However, it is not obvious how to provide the necessary microarchitectural support, and few details have been given in the literature. By judiciously constraining microthreads, we can easily adapt a superscalar machine to support many simultaneous microthreads. The nature of precomputation microthreads also requires efficient usage of resources. Our proposed implementation addresses this issue by dynamically identifying and aborting useless microthreads.

1. Introduction

The use of *precomputation microthreads* has recently become a popular topic of architecture research. Several recent publications have demonstrated significant potential to improve performance [2, 3, 7, 9]. Precomputation microthreads are light-weight threads that execute data flow captured from the primary thread in order to compute useful information. For example, branch conditions and load addresses can be precomputed to benefit branch prediction and prefetching.

Providing support for precomputation microthreads is not trivial, nor is it obvious. Microthreads execute concurrently with the primary thread and must share execution resources. The microarchitecture should ensure that the primary thread is minimally impacted by the presence of microthreads. At the same time, the microarchitecture must ensure that microthreads receive enough execution bandwidth to improve performance. Previous papers have not dealt with implementation concerns in detail.

The presence of *false spawns* further complicates the design problem. False spawns launch microthreads that do not produce useful results due to control-flow mismatches with the primary thread. Our analysis shows that the problem of false spawns must be addressed, but that it cannot be easily avoided algorithmically.

This paper describes a superscalar machine that is augmented for support of precomputation microthreads. We propose four microthread constraints and show how they enable a straightforward design without greatly compromising functionality. Furthermore, our design promotes efficient use of the hardware by identifying false spawns and aborting the resultant useless microthreads.

This paper is organized into six sections. Section 2 discusses prior research relevant to this paper. Section 3 describes the constraints we place on microthreads and rationale for each. Section 4 describes our superscalar core adapted to support microthreads. Section 5 describes how to promote efficiency by aborting microthreads launched by false spawns. Section 6 provides experimental analysis. Section 7 provides conclusions.

2. Related Work

Several previous works have proposed the use of simultaneous precomputation microthreads. Implementation and efficiency issues were addressed in some of these papers, but not treated in great detail.

Chappell *et al.* proposed the Simultaneous Subordinate Microthreading (SSMT) paradigm as a general method for leveraging spare execution capacity to benefit the primary thread [1]. Subsequent authors have referred to subordinate microthreads also as "helper threads." The paper described general hardware support for *microcontexts*, the state associated with running microthreads. However, they did not provide implementation details or address live-in communication needed for precomputation microthreads.

Chappell *et al.* also proposed the use of precomputation microthreads to improve branch prediction along difficult paths [2]. (The desirability of precomputing only hard to predict branches was noted by Uri Weiser in 1995 [8].) Their paper focused on hardware support for microthread generation, rather than general support for microthreads. A mechanism to abort useless microthreads was mentioned, but few details were provided. Our paper addresses that

mechanism in greater detail.

Roth and Sohi used precomputation microthreads in the Data-Driven Multithreading paradigm [7]. Microthreads were triggered by fetch PC, the primary thread register map was copied to support live-in values, and efficiency was promoted by delaying spawns until they became less speculative. Some microthread constraints in their mechanism matched what we have concluded: microthreads should not contain branches and should be allowed to retire out-of-order.

Zilles and Sohi proposed using precomputation microthreads called *speculative slices* [9]. Speculative slices were hand-constructed and were executed on unused contexts of a simultaneous multithreading machine. Microthreads received live-in register mappings by borrowing renamer ports, though it was not clear how this was implemented. Zilles and Sohi proposed a mechanism similar in spirit to an abort mechanism to maintain their prediction correlator. "Kill" instructions were inserted into the primary thread to signal when its control-flow path deviated from that expected by the microthread.

Moshovos *et al.* proposed precomputation microthreads for prefetching using a Slice Processor [6]. Although the implementation in the paper provided separate *scout units* to execute microthreads, they also suggested running microthreads on spare SMT contexts. As in [7], the Register Alias Table of the primary thread was copied to map live-in values. Efficient use of the available resources was not addressed, nor was it as critical, as they provided dedicated resources.

Collins *et al.* proposed using precomputation microthreads to generate prefetches as part of the Speculative Precomputation paradigm [3, 4]. In this scheme, microthreads were run on spare contexts of an SMT machine. The authors proposed a method of communicating live-ins in which MOVE instructions were inserted into the primary thread's instruction stream. Implementation of this was not discussed.

3. Microthread Constraints

To limit design complexity and the impact of microthread overhead, microthread support should be added with as few changes to the existing hardware as possible. This goal can be encouraged by adopting a set of simplifying constraints for microthreads. We propose the following constraints:

- **No explicit control flow**. Microthread branches would increase microthread length and complicate microthread construction. They would also require branch prediction/recovery hardware or, alternatively, would require microthreads to wait for branches to resolve. Branch prediction within microthreads raises the possibility of mispredictions and the necessity of recoverable state. Stalling microthreads to wait for branch resolution increases latency and stalls the front-end pipeline.

- **No live-out registers**. If live-out registers (those with life outside the current microthread invocation) are prohibited, no hardware is required to hold persistent microthread register values, and there is no need to retire microthread instructions in order. This saves hardware cost/complexity and increases the rate at which microthread instructions can be removed from the machine. If the microthread algorithm requires persistent state across invocations, it can be stored in memory.

- **Shared memory context**. Precomputation microthreads frequently perform loads only from the primary thread's memory context, but no existing precomputation mechanisms execute microthread stores. By always sharing the primary thread's memory context, we eliminate the need to modify the memory system to support separate contexts for microthreads. However, if the baseline core already supports SMT, additional memory contexts could be used to support microthreads. We do not explore this possibility.

- **Speculative results**. If microthreads are always viewed as speculative, their results can be discarded. This allows design flexibility within the core and aggressive control mechanisms, such as the abort mechanism described in Section 5. If microthreads are not speculative, the implementation must guarantee correctness, which adds design complexity and limits aggressive speculation.

4. Adapting the Processor Core

Unlike most previous papers, our implementation does not assume or require SMT capability. As such, we describe our support for precomputation microthreads using a conventional superscalar out-of-order machine as a baseline.

A high-level diagram of our adapted machine is shown in Figure 1. The unshaded area depicts the baseline components. Primary thread instructions are fetched from either an instruction cache or trace cache, decoded, renamed according to the Register Alias Table (RAT), and then issued into the reservation stations. When an instruction's sources have been produced, that instruction is scheduled for execution on a functional unit. Instructions are retired in-order as they complete.

The shaded area of Figure 1 depicts new hardware that is specific to the processing of microthreads. The remainder of this section describes the shaded hardware and describes how some primary thread structures, such as the Register

Alias Table, must be modified from their original implementation.

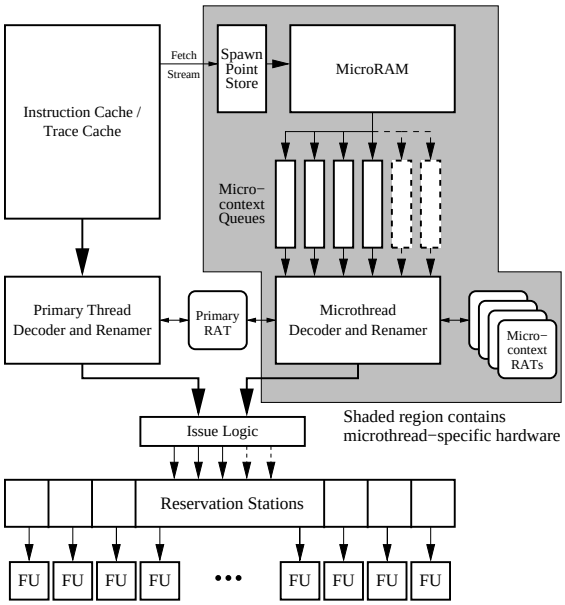

Figure 1. Superscalar Processor Core.

4.1. Microthread Fetch Pipeline

Microthread fetch requirements are very different from those of the primary thread. Microthreads are small and can be stored in their entireties on-chip, eliminating the possibility of fetch breaks due to cache misses. Since we have constrained microthreads not to include branches, there is no need for branch predictions or fetch redirections.

Because microthread fetch is different from primary thread fetch, it makes sense to separate the pipelines almost entirely. This keeps both fetch pipelines focused, possibly at the expense of additional hardware. Since hardware cost is no longer as critical as design complexity and hardware placement, we believe this to be an appropriate trade-off.

The microthread fetch pipeline, shown in detail in Figure 2, consists of the *Spawn Point Store*, the *MicroRAM*, and a set of *Microcontext Queues* (shown in Figure 1).

The only requirement of the primary thread fetch engine is to provide its stream of fetch addresses to the Spawn Point Store.

4.1.1. Spawn Point Store. The Spawn Point Store is a mapping between primary thread fetches and MicroRAM entries. Each fetch of the primary thread indexes an entry in the Spawn Point Store, which provides the addresses of all the microthreads to be spawned by that fetch. The Spawn Point Store is logically an extension of the primary thread's instruction caching structure, and thus the two could be integrated. In this paper, we assume it to be a separate, parallel structure. Each fetch of the primary thread accesses up to four MicroRAM addresses. In practice, the number of

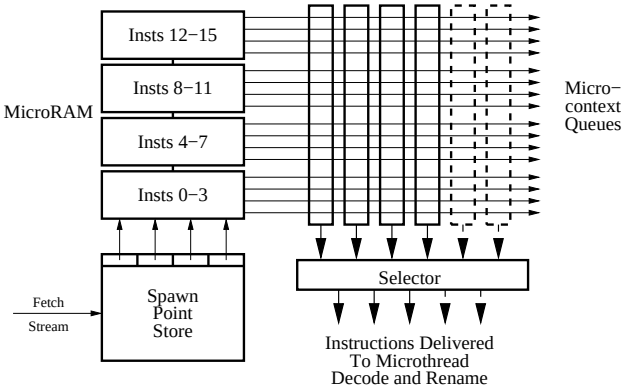

Figure 2. Microthread Fetch Pipeline. Only one set of MicroRAM output wires is shown.

Microcontext Queues limits microthread bandwidth more quickly than the number of MicroRAM addresses stored per entry in the Spawn Point Store.

4.1.2. MicroRAM. Each MicroRAM entry contains an entire microthread routine. We assume that there are 1024 MicroRAM entries, and that each entry can hold a routine of up to 16 instructions. The number of ports required for each MicroRAM cell is equal to the maximum number of addresses provided by the Spawn Point Store (in our case, four). In general, the MicroRAM structure can be built quite large (if hardware budget allows), since it is easily pipelined by dividing the memory array such that a fetch ripples across the word lines. For example, a 16 instruction fetch could be divided into four fetches of four instructions, as shown in Figure 2.

4.1.3. Microcontext Queues. Instructions fetched from the MicroRAM are deposited into one of several Microcontext Queues. Each Microcontext Queue holds up to an entire microthread routine (16 instructions). A queue is allocated before the MicroRAM access begins, and the Microcontext Queue number is passed along with the instructions coming out each port of the MicroRAM such that the proper Microcontext Queue can conditionally latch the instructions. Each cycle, a number of microthread instructions are selected in round-robin fashion from the fronts of the queues and sent to the microthread decoding and renaming stages. For simplicity, each queue can contain only one microthread at a time. A queue is deallocated when all instructions from the microthread have been removed.

4.2. Microthread Decode and Rename Pipeline

4.2.1. Microthread Instruction Decoders. The decoders operate on instructions coming from Microcontext Queues. It is assumed they are the equivalent of micro-ops—instructions decoded into the internal ISA of the machine. Decoding them should require few cycles, if any.

We provide fewer decoders than Microcontext Queues, since it is unlikely that all of the queues simultaneously have instructions ready to issue. Figure 2 shows a selector that chooses a subset of instructions to remove from Microcontext Queues and send to the decoders.

Precomputation trees tend to have low instruction-level parallelism, but there may be enough that microthreads can benefit from superscalar issue. We investigate allowing multiple instructions to be removed from a single queue in our experiments. In these cases, the selector will choose instructions from other queues before a second instruction from any single queue.

4.2.2. Microthread Register Renamers. The constraints discussed in Section 3 simplify microthread register renaming. Logical-to-physical register tag mappings do not exist outside the invocation of a microthread, eliminating the need for persistent state and localizing Register Alias Table information. Furthermore, RAT state does not need to be checkpointed in anticipation of a branch recovery.

Microthread renaming hardware is further simplified because few instructions from one microthread need be renamed in a single cycle. It is not necessary to "ripple-rename" many dependent instructions together, as the primary thread renamer must do. The number of microthread instructions that can be ripple-renamed is determined by the number of instructions that can be removed from a Microcontext Queue in a single cycle (discussed in previous section). We experiment with this parameter in our experiments.

Microthread renamers allocate physical register tags for output registers. The global register tag allocation mechanism must be modified to include this capability. We do not believe this to be a significant change.

4.2.3. Live-Input Renaming. Precomputation requires live-input values from primary thread registers. Previous works accomplished live-input renaming in multiple ways, as described in the Related Work section. All of these methods have drawbacks, and implementations were not addressed in detail.

Our implementation provides live-input values by renaming microthread live-input registers such that they receive the appropriate physical register tags from the primary thread. After the renaming phase, microthread instructions naturally source the appropriate values. This approach was suggested in [7], [9], [6], and [2]

Microthread instructions must be renamed according to the primary thread's Register Alias Table contents *at spawn time*. Copying the Register Alias Table could solve this problem, but it is not desirable: it does not scale well with the number of active microthreads and complicates the primary thread's hardware. Inserting MOVE instructions, as proposed in [3], does not eliminate the problem: the move

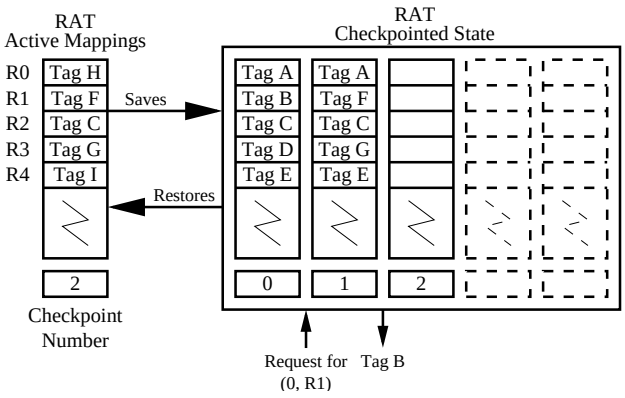

Figure 3. Adapted Primary Thread RAT

instructions themselves must be renamed at the point of the spawn. Furthermore, the additional instructions waste issue and execution bandwidth, especially as the number of active microthreads is scaled up.

Our method of live-input renaming is based on the following two observations. First, it is very difficult to rename microthread instructions before the primary thread's spawn-causing instruction, since the primary thread's RAT mappings at that point are needed. Second, a modern machine checkpoints the contents of the RAT frequently in order to prepare for possible branch misprediction recoveries. These two properties ensure that the RAT content necessary to rename microthread live-input registers already exists within the RAT's checkpointed state. All that is needed is a means to access it.

We provide read ports to access the checkpointed state. The exact implementation depends on the existing RAT hardware. An example is shown in Figure 3. Note that the read ports need not provide access to the *active* RAT state, since it is guaranteed that a microthread will be renamed at least one cycle after its spawn-causing instruction. Since the checkpointed state serves no purpose other than to store mappings for recovery, we believe adding read ports is not difficult.

Our implementation assumes that spawn-causing instructions force the RAT to be checkpointed as does a branch. Alternatively, spawn-causing instructions could be restricted such that they must fall on checkpoint boundaries. In either case, checkpoint numbers are stored and associated with the appropriate microthreads as they are spawned.

A checkpointed RAT read port is not necessary for every microthread renamer, since the number of live-input registers is relatively small for typical precomputation trees. The ports provided are arbitrated, stalling whenever live-input renaming beyond the provided ports is required. We investigate the number of ports required in our experimental section.

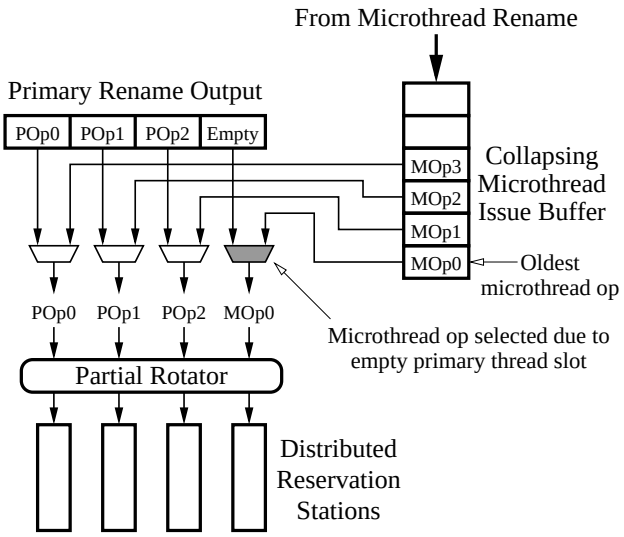

Figure 4. Issue Logic Into The Reservation Stations. Empty slots are filled with the oldest microthread instructions waiting to issue.

4.3. Issue Logic

The issue logic, shown in Figure 1 (for a 4-wide machine), sends renamed instructions to the machine's reservation stations. The reservation stations in our model are designed to support a 16-wide superscalar machine. There are 16 distributed sets, each with its own scheduler choosing from 32 instruction entries. Instructions are issued "flat" across the reservation stations, such that each set gets at most one new instruction per cycle. The issue logic contains a partial rotator to spread instructions evenly across all 16 sets.

To support microthread issue, each rotator slot is adapted to accept input from either the primary thread rename stage or the microthread rename stage, as shown in Figure 4. The microthread front-end pre-aligns its instructions using a collapsing buffer to eliminate the need for a routing crossbar. The result is a single 2:1 multiplexer for each instruction issue slot. Each multiplexer selects the primary thread input unless that issue slot is empty, in which case it selects the microthread input.

Since each multiplexer is very small and can be selected cycles in advance, the additional stage delay incurred by this design is minimal (a pass gate or transmission gate delay). We do not believe this will influence the cycle time of the machine.

We treat full issue cycles of the primary thread as a special case, since we do not want microthreads to starve. Immediately following any cycle in which the primary thread uses the full bandwidth, the issue logic performs a microthread-only issue cycle. This condition is detected in advance and a one-cycle bubble inserted, causing the multiplexers automatically to select up to the maximum number of microthread instructions. If no microthread instructions are waiting to issue, the bubble is not inserted.

An alternative to our scheme is to modify each set of reservation stations to accept up to two instructions per cycle: one primary thread and one microthread. In such an implementation, issue bandwidth is effectively doubled, though it requires twice as many routing busses and write ports. We compare our implementation to this optimistic alternative in our experiments section.

Another alternative might be to select between full-issue cycles of the primary thread and microthreads using a thread selection policy. Our per-instruction approach is superior to this alternative. It makes better use of the available issue bandwidth, properly favors the primary thread, and is less likely to lengthen critical dependency chains. We do not provide experimental results for this comparison.

4.4. Execution and Retirement

The machine's execution core and retirement hardware are almost completely unaffected by the presence of microthreads. Once microthread instructions are issued into the reservation stations, they behave just like instructions from the primary thread. The only modification necessary is that the execution core allow microthread instructions to simply "fall off the end" of the execution pipeline, rather than be processed by the in-order retirement hardware.

Key hardware structures are unaffected by the presence of microthread instructions. Most importantly, the register files, reorder buffer (or equivalent), and memory system are unchanged. Since microthreads have no register state outside of an invocation, no register file space is needed to hold architectural register values. Because microthread instructions can retire out-of-order, reorder buffer hardware is not necessary. Because microthreads share memory context with the primary thread, memory instructions are processed normally (though we suppress all memory exceptions, such as TLB misses, caused by microthread instructions). Because microthreads are speculative, we do not need to enforce memory ordering strictly.

4.5. Scaling the Adapted Core

The adapted processor core scales well with the number of active microthreads allowed, which benefits algorithms that rely on high microthread bandwidth. Each additional microthread supported adds another Microcontext Queue and another input into the decode selector. If several are added, it may be desirable to add a microthread decoder/renamer slot. It may also be desirable to add another checkpointed RAT read port to support more live-inputs per cycle. The issue logic itself is unaffected by scaling, since bandwidth is fixed to the overall issue width of the machine. However, the issue logic could be modified to provide more

microthread issue bandwidth by intermittently performing microthread-only issue cycles.

It should be noted that this type of scalability is not available in implementations that rely upon spare SMT contexts to execute microthreads, as additional active microthreads cannot be supported without adding full SMT contexts. For this and similar reasons, we find such SMT-based approaches to be unnecessarily restrictive. Microthreads do not have the same goals or require the same support as SMT threads. We consider microthread support and SMT support to be mostly orthogonal issues.

5. Promoting Efficiency

Efficient use of resources is necessary for precomputation microthreads to be successful. Because microthreads contend for resources with the primary thread, it is important to minimize the number of microthreads that do not perform useful computations.

Phenomena we call *false spawns* launch useless microthreads. This section describes the problem of false spawns and how our implementation combats their negative effects by quickly aborting the resulting useless microthreads.

5.1. False Spawns

Each precomputation microthread executes data flow from the primary thread along a particular control-flow path. If the primary thread deviates from the path assumed by a microthread, that microthread does not produce a useful value[1]. These microthreads are said to be launched by false spawns.

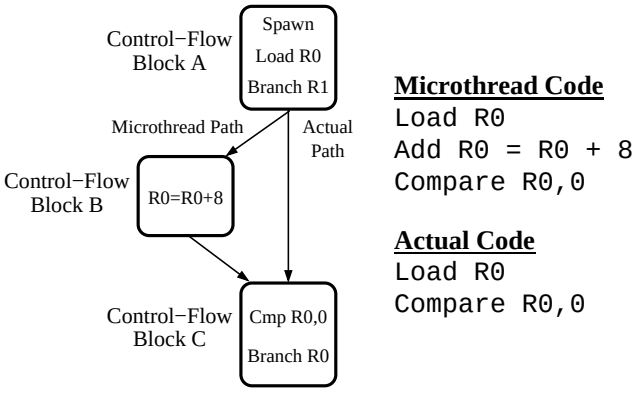

Figure 5. A False Spawn

Figure 5 demonstrates the concept of a false spawn. The spawn in block A launches a microthread to precompute the condition of the branch in block C. The microthread code assumes the primary thread will follow the control-flow path ABC. However, the primary thread actually takes the path AC, skipping the ADD instruction in block B. The precomputed result would be discarded, since the condition computed is likely incorrect. The spawn in block A is a false spawn.

False spawns are damaging because the microthreads they launch consume resources without performing useful work. Not only do false spawns cause unnecessary contention with the primary thread, they may prevent other, potentially useful microthreads from executing.

In general, false spawns become more numerous as the depth of speculation increases. Each time a spawn point is hoisted into a new basic block, there is a higher chance that the actual control-flow path will deviate. In an aggressive, 16-wide machine, deep speculation is necessary to keep microthreads precomputing ahead of the primary thread. As such, spawn points are often hoisted ten or more basic blocks, yielding many false spawns.

The problem of false spawns was first noted in [7]. They combated the problem by delaying spawns as much as possible, in effect waiting until spawn points became less speculative. At the depth of speculation we study, this approach has little impact and also negatively effects microthread latency.

5.2. Choosing Better Spawn Points

One possible solution to the false spawn problem is to select better spawn points for microthreads. By nature, programs have many convergent control-flow points. If spawn points could be selected such that they are close to control-equivalent with the target point in the primary thread, perhaps false spawns could be reduced to the point that they are tolerable.

We investigate the potential of spawn point selection in the experimental section of this paper (Section 6). Our results demonstrate that, in general, is not possible to find spawn points that limit false spawns while maintaining a high depth of speculation.

5.3. Aborting Useless Microthreads

Our implementation uses a dynamic mechanism to detect and abort microthreads launched by false spawns. By associating a control-flow signature, called $Path\_History$, with each running microthread, we can detect when the primary thread fetches down an alternative control-flow path. As soon as a deviation is detected, we can abort the running microthread, limiting the negative effects of the false spawn.

5.3.1. Abort Detection. $Path\_History$ is a bit vector that encapsulates the control-flow path represented by each microthread. It is a concatenation of a few bits from the instruction address of each taken control-flow instruction covered by the microthread. The vector is constructed at the same time as the microthread, whether at compile-time

[1] Microthreads launched by false spawns can still have useful side-effects, such as prefetching memory values.

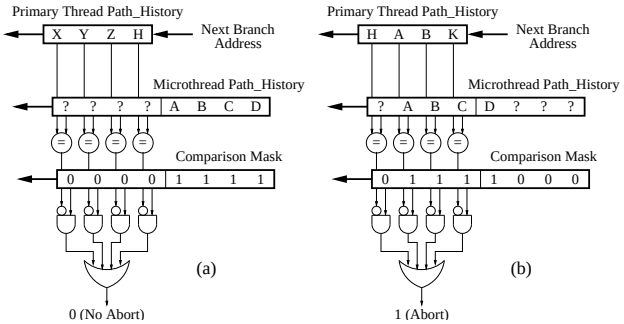

Figure 6. Abort Detection: (a) State at spawn time, (b) Mismatch on third partial address signals an abort.

or run-time, and is stored in the MicroRAM. When a microthread is spawned, its $Path\_History$ vector is copied into an active list.

The primary thread's front-end maintains its own $Path\_History$ vector based on the control-flow instructions it fetches. Bits from each partial address are shifted in as control-flow changes occur.

Each cycle, the primary thread's current $Path\_History$ vector is compared to part of the $Path\_History$ entry for each active microthread. Comparisons are controlled by a comparison bit mask. Each microthread's $Path\_History$ vector and comparison bit mask are left-shifted each time the primary thread's $Path\_History$ shifts in a new partial address. This ensures that the comparison occurs on the correct blocks.

If a mismatch occurs, the primary thread has fetched down a path that differs from the microthread and an abort is signaled. Figure 6 shows how an abort is signaled. The state of the mechanism immediately following a spawn of a microthread is shown in Figure 6(a). A mismatch case is shown in Figure 6(b). The microthread expects path $ABCD$, but mismatches on the third address (K instead of C). Note that all of the comparators in the figure may not be necessary, depending on the maximum number of control flows that can processed in a single cycle.

Branch mispredictions affect the $Path\_History$ vectors used by the abort mechanism. If a misprediction occurs, the $Path\_History$ registers are shifted in the opposite direction to recover to the point of the misprediction. This is implemented by making each shift register long enough to hold partial addresses for every unresolved branch, preventing any loss of state.

5.3.2. Limitations of an Abort. The abort mechanism is limited to flushing instructions that are still in the Microcontext Queues. Microthread instructions cannot be flushed once they enter the reservation stations of the execution core. We assume that partially flushing microthread instructions from the out-of-order window is too complex to implement reasonably.

5.4. Abort Mechanism Implementation

The implementation of the abort mechanism is straightforward. Maintaining the $Path\_History$ in the primary thread front-end is trivial: a shift register combines a few bits of each taken branch address. The active list of microthread $Path\_History$ vectors has few entries (we assume 16), so the comparison logic shown in Figure 6 need not be duplicated excessively. As previously mentioned, we do not flush microthread instructions once they enter the reservation stations, so there is no additional complexity outside of the microthread front-end.

5.5. False Abort Problems

Our abort mechanism is limited by the speculative nature of the primary thread's front-end. It is always possible that the machine is proceeding down an incorrect control-flow path due to a branch misprediction. Given this, when our abort mechanism detects a mismatch and signals an abort, it may be a *false abort*.

A false abort may or may not be harmful. If a false abort occurs but the primary thread still does not take the path expected by the microthread, then the false abort had the same effect as a normal abort. However, if a false abort occurs for microthread precomputing along the true path of execution, then the potential gain of that microthread is lost.

A *false no-abort* may also harm performance. In this case, a microthread is *not* aborted because of a mispredicted branch. This has the same effect as if there were no abort mechanism at all: false spawns inject useless instructions into the machine, possibly impacting the primary thread and possibly preventing other microthreads from spawning.

In general, false aborts are more harmful than false no-aborts. When a false no-abort occurs, the machine will eventually recover and the abort mechanism will have a chance to correctly abort the microthread, albeit later than would have occurred normally. On the other hand, when a false abort occurs, the positive potential of the microthread is lost forever.

Avoiding False Aborts. We avoid false aborts by using branch confidence [5] to estimate whether or not the primary thread is fetching down the correct path. For each partial address in the front-end's $Path\_History$ vector, a branch confidence bit is attached. Microthread aborts are suppressed unless all addresses being compared are marked as confident up to and including the mismatching address. In the mismatch case shown in Figure 6(b), addresses A, B, and K would need to be marked confident in order for the abort to proceed. The confidence mechanism is biased such that aborts that do occur are highly likely to be legitimate.

This modified mechanism is not perfect. Branch confidence mechanisms are not always accurate, and nothing is being done to prevent false no-aborts. Our future work includes experimenting with a method of stopping

Table 1. Baseline Machine Model

Fetch, Decode, Rename	64KB, 4-way associative, instruction cache with 3 cycle latency capable of processing 3 accesses per cycle; 16-wide decoder with 1 cycle latency; 16-wide renamer with 4 cycle latency
Branch Predictors	128K-entry gshare/PAs hybrid with 64K-entry hybrid selector; 4K-entry branch target buffer; 32-entry call/return stack; 64K-entry target cache (for indirect branches); all predictors capable of generating 3 predictions per cycle; total misprediction penalty is 20 cycles
Execution Core	512-entry out-of-order window; physical register file has 4 cycle latency; 16 all-purpose functional units, fully-pipelined except for FP divide; full forwarding network; memory accesses scheduled using a perfect dependency predictor
Data Caches	64KB, 2-way assoc L1 data cache with 3 cycle latency; 4 L1 cache read ports, 1 L2 write port, 8 L1 cache banks; 32-entry store/write-combining buffer; stores are sent directly to the L2 and invalidated in the L1; 64B-wide, full-speed L1/L2 bus; 1MB, 8-way associative L2 data cache with 6 cycle latency once access starts, 2 L2 read ports, 1 L2 write port, 8 L2 banks; caches use LRU replacement; all intermediate queues and traffic are modeled
Busses and Memory	memory controller on chip; 16 outstanding misses to memory; 32B-wide core to memory bus at 2:1 bus ratio; split address/data busses; 1 cycle bus arbitration; 100 cycle DRAM part access latency once access starts, 32 DRAM banks; all intermediate queues modeled

microthreads, rather than completely aborting them. This would allow the possibility of restarting them later.

6. Experimental Analysis

This section analyzes the design decisions proposed in Sections 4 and 5. We experimented with a hypothetical machine that used our hardware implementation model to execute microthreads generated by the difficult-path branch prediction algorithm published in [2].

It is important to experiment with a successful microthread-based mechanism for several reasons. Many of our design decisions place constraints upon the capabilities of the microthreads—we must ensure that performance potential is not greatly compromised. It is impossible to test this claim without a successful algorithm for creating useful microthreads. The experimental results in this section demonstrate that it is possible to constrain microthreads as we have suggested, without impairing the ability of a successful mechanism to improve performance.

Although it was necessary to choose a particular microthread application and algorithm to generate experimental results, we believe the contributions of this paper can be generalized to apply to other mechanisms using precomputation microthreads, such those listed in the Related Work section. The implementation concerns we address apply to any microthread-based machine, independent of what the microthreads are actually doing. The path-deviation problem we address (which results in false spawns) exists in any mechanism that performs speculative precomputation.

6.1. Baseline Machine Model

Our baseline configuration for these experiments was an aggressive, wide-issue superscalar machine. The machine parameters are summarized in Table 1. All experiments were performed using the SPECint2000 benchmark suite compiled for the Alpha EV6 ISA with `-fast` optimizations and profiling feedback enabled.

Our machine used an idealized front-end to avoid biasing our results. Microthreads take advantage of resources unused by the primary thread. A fetch bottleneck would unfairly under-utilize execution resources, leaving more for the microthreads to consume and lessening the negative impact of microthread overhead. Our front-end can handle three branch predictions and three accesses to the instruction cache per cycle. In a sense, we are modeling a very efficient trace cache.

6.2. Adapted Machine Model

We altered our baseline machine model to support microthreads, based on the approach discussed in Section 4. Our experiments investigated the effects of varying multiple configuration parameters. All performance results are shown as speedup versus the baseline machine model.

For each experiment, we simulated two basic microthreaded models: unlimited and limited. The unlimited version was intended to show our implementation without constraints. The limited version was intended to be a realistic configuration. When a parameter was varied, the remaining parameters were set according to Table 2.

Table 2. Microthreaded Machine Models

	Unlimited	Limited
Effective Overall Issue Width	16	16
Microcontext Queues	16	16
Microthread Decoders/Renamers	16	8
Issue Width Per Microcontext Queue	16	2
Live-Input Rename Ports	16	2

6.3. Implementation Sensitivity Experiments

The experimental results in this section demonstrate that the parameters we have chosen for our Limited configuration do not significantly impact the ability of the microthread algorithm to improve performance.

All experiments in this section use the abort mechanism presented in Section 5. The effects of the abort mechanism itself are studied in Section 6.4.

6.3.1. Microthread Issue Bandwidth. Our implementation shares the overall issue bandwidth of the machine between the primary thread and microthreads. This is simpler to implement than providing double the routing busses and write ports into the reservation stations.

Figure 7 shows the performance of our two basic configurations using shared bandwidth (16-wide) versus implementations that use double issue bandwidth (32-wide). On average, there was little difference between the two. For some benchmarks, the additional bandwidth was beneficial, as more microthreads were allowed to complete with

lower latency. However, in other benchmarks, the additional bandwidth led to increased microthread overhead and slightly lower performance. In summary, our shared approach sacrifices a little potential when compared to double bandwidth, but the small difference is more than made up by the simpler design.

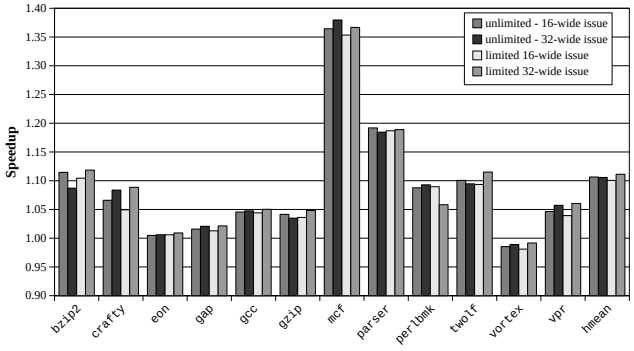

Figure 7. Shared Issue Bandwidth Versus Double Issue Bandwidth

6.3.2. Microthread Decode/Rename Slots. Our implementation assumes that a fixed number of microthread instructions can be decoded per cycle. Since decoded instructions are fed directly into the microthread renamers, the number of renamers equals the number of decoders. We also assume that each microthread renamer is capable of allocating a physical register tag. We varied all three of these implementation parameters together.

The performance results are shown in Figure 8. Since all 16 Microcontext Queues are unlikely to be ready to issue every cycle, it was not necessary to provide 16 microthread decode/rename slots. Eight, twelve, or even four slots provided enough bandwidth without constraining microthreads too heavily. As in the previous experiment, the additional microthread bandwidth was not always helpful for every benchmark, since microthread overhead can increase as more bandwidth is provided.

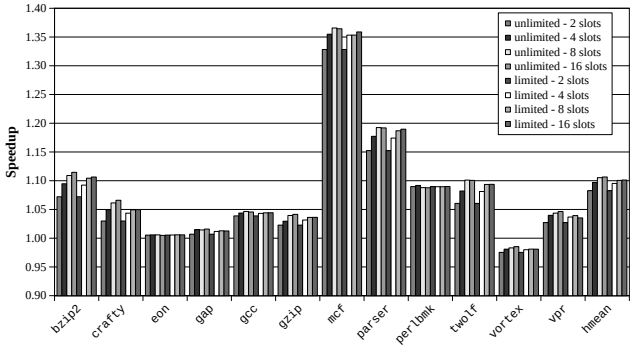

Figure 8. Varying Microthread Decode and Rename Slots

6.3.3. Issue Width Per Microcontext Queue. Our implementation assumes that each Microcontext Queue can issue a limited number of instructions per cycle. If precomputation microthreads contain mostly long chains of dependent instructions, it should be possible to limit issue bandwidth per Microcontext Queue without significantly impacting performance. As discussed in Section 5, this is desirable because it simplifies microthread renamer design.

Figure 9 shows performance speedup as issue width per Microcontext Queue was varied. The results show that additional issue width per queue was not tremendously beneficial past two or four instructions per cycle. In fact, it was sometimes worse, as simultaneous issue of likely dependent operations takes up space in the out-of-order window without providing additional instruction-level parallelism.

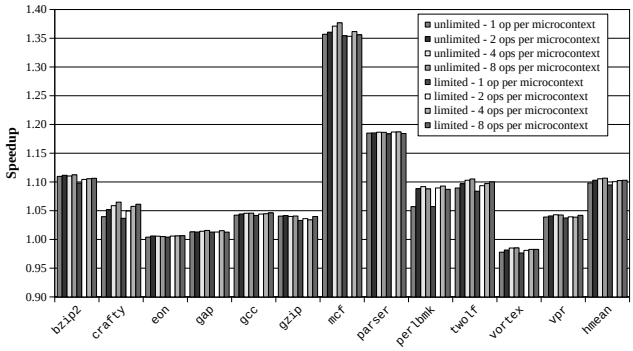

Figure 9. Varying Issue Width Per Microcontext Queue

6.3.4. Live-Input Rename Ports. Our implementation provides read ports from the checkpointed RAT state to rename microthread live-input registers. These ports are shared among all microcontext renamers. Of course, we would like to minimize the number ports.

The results in Figure 10 show that there was little benefit to providing more than two or four live-input rename ports. This is what one would expect, since only a subset of precomputation microthread instructions source register values from the primary thread.

6.3.5. Scalability of the Number of Microcontexts. A major advantage of our implementation is its scalability. Some microthread mechanisms perform better with numerous microthreads in-flight simultaneously, such as the one with which we experiment. These mechanisms benefit from an implementation capable of supporting many in-flight microthreads.

To illustrate the potential benefit, we measured performance with a variable number of Microcontext Queues. The results are shown in Figure 11. Clearly, the algorithm was able to take advantage of the additional capacity, gaining about 5% performance on average. This is because many

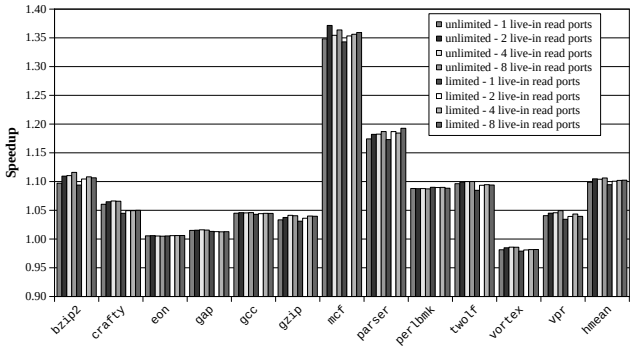

Figure 10. Varying Microthread Live-Input Rename Ports

highly-speculative microthreads are launched (though most are aborted), requiring high microthread bandwidth.

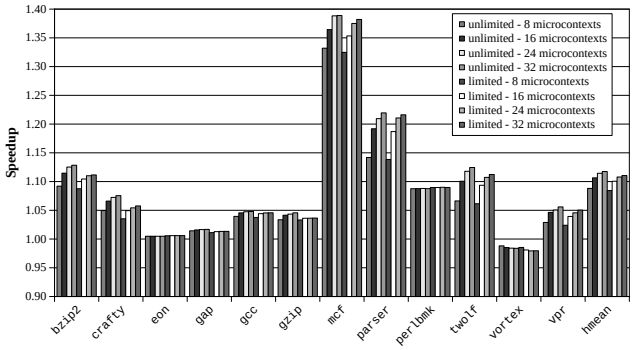

Figure 11. Varying Number of Microcontexts

6.4. Microthread Efficiency Experiments

6.4.1. The Basic Abort Mechanism. Figure 12 shows the performance impact of disabling our abort mechanism. In the majority of benchmarks, disabling the abort mechanism saturates the machine's Microcontext Queues with junk microthreads caused by false spawns. These microthreads interfere with the primary thread, but do not remove branch mispredictions, all but destroying the performance gain. If it were not for the second-order prefetching effects caused by the falsely-spawned microthreads, the performance loss would be much greater.

Perlbmk is unusual. It especially benefits from false spawn prefetching effects. Further, perlbmk has a relatively low number of false spawns, lessening the importance of the abort mechanism. Because of these two factors together, disabling the abort mechanism actually increased performance slightly.

6.4.2. Spawn Utilization. It was mentioned in Section 5.2 that a possible solution to the false spawn problem is to choose better spawn points, such that microthreads launched have a higher probability of precomputing useful values.

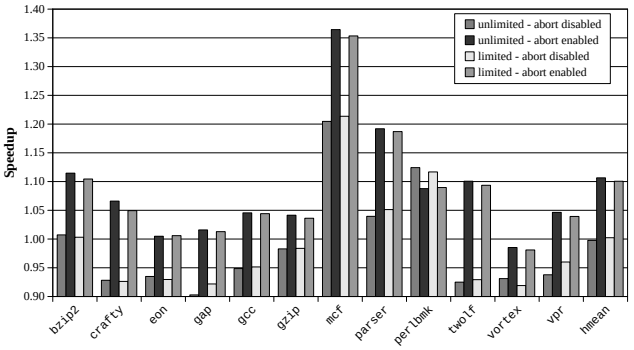

Figure 12. Disabling The Abort Mechanism

Figure 13 shows that, on average, it is not possible to find good spawn points. In this experiment, we investigated several depths of speculation, including 2, 4, 6, 8, and 10 control-flow changes (taken branches). For each path (of the given depth) to a branch, we considered all instructions on that path as possible spawn points. Figure 13 shows the average utilization of the best spawn points chosen. Utilization is simply the number of times the spawn instruction led to the branch along the given path divided by the total number of times the instruction was executed. Low utilization means many false spawns.

The existence of good spawn points is determined by benchmark control-flow structure and tendencies during execution. Sometimes, increasing the depth of speculation opens up opportunities to select better spawn instructions. This occurred for some depths in perlbmk, vortex, and gap. Much more often, however, increasing the depth yielded worse spawn points. Higher speculation depth creates more paths that are taken less frequently, making it generally much more difficult to find a good spawn point for each path.

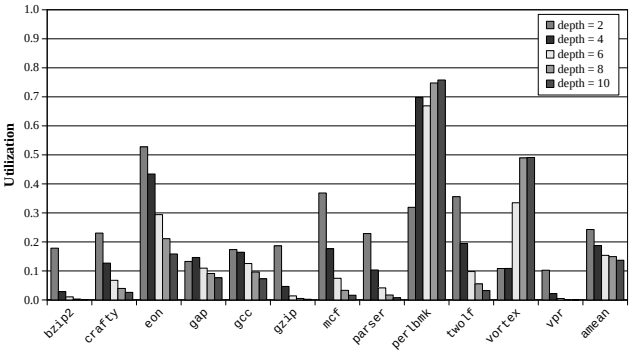

Figure 13. Average Ratio of Useful Spawns to Overall Spawns

6.4.3. Handling False Aborts. False aborts, described in Section 5.5, are a problem due to the speculative progress of the primary thread. False aborts eliminate performance potential by falsely eliminating useful microthreads. To address this problem, we modified our abort mechanism to use

branch confidence to mask low-confidence aborts.

The performance impact of false aborts is shown in Figure 14 as the difference between the "perfect" and "real" bars. On average, using a real confidence mechanism achieved half the performance improvement of using a perfect confidence mechanism. This is because our real-confidence abort mechanism is very careful to avoid false aborts, thereby increasing the number of false no-aborts. Clearly, our solution can be further improved, but we have successfully reclaimed over half of the lost performance.

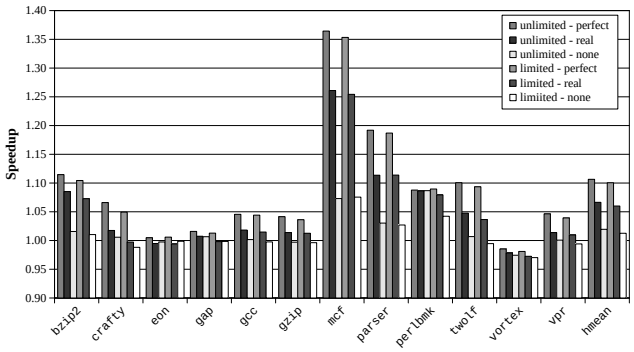

Figure 14. Using Perfect, Real, and No Branch Confidence to Guide Aborts

7. Conclusions

Precomputation microthreads have been shown to have a great deal of performance potential. Several previous studies have investigated the use of precomputation microthreads for improving prefetching and branch prediction. However, none of these works provide significant implementation details.

This paper proposes an implementation for precomputation microthreads designed to minimally impact an existing superscalar machine. By judiciously constraining microthreads, we can provide a simple, scalable core design that supports many simultaneous microthreads. Our experimental results demonstrate that our implementation does not significantly impact the performance potential of a successful precomputation microthread-based mechanism.

Our implementation also addresses microthread efficiency by dealing with the problem of false spawns. False spawns limit performance by causing unnecessary contention with the primary thread and by consuming resources that could be used by other microthreads. Since the problem cannot be solved satisfactorily by selecting better spawn points, our mechanism reacts by identifying and aborting microthreads caused by false spawns. Though our abort mechanism performs well, its utility is still limited by false aborts and false no-aborts. Our future work involves solving this problem by pausing and possibly restarting microthreads, rather than aborting them irrevocably.

Finally, we believe the concepts and results presented in this paper are applicable to any mechanism using precomputation microthreads. The key challenges we address—fetching, decoding, renaming, and issuing microthreads; live-input register renaming; and eliminating the effects of false spawns—are applicable to general microthreading and precomputation mechanisms.

8. Acknowledgments

Robert Chappell is a Michigan PhD student on an extended visit at The University of Texas at Austin. We gratefully acknowledge the Cockrell Foundation and Intel Corporation for his support. Francis Tseng's stipend is provided by an Intel fellowship. We also thank Intel for their continuing financial support and for providing most of the computing resources we enjoy at Texas. Finally, we are constantly mindful of the importance of our regular interaction with the other members of the HPS group and our associated research scientists.

References

[1] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 186 – 195, 1999.

[2] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. Difficult-path branch prediction using subordinate microthreads. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 307 – 317, 2002.

[3] J. Collins, D. M. Tullsen, H. Wang, and J. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[4] J. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[5] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 142–152, 1996.

[6] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the 2001 International Conference on Supercomputing*, 2001.

[7] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture*, 2001.

[8] U. Weiser. Personal Communication, 1995.

[9] C. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.