The Use of Multithreading for Exception Handling

Craig B. Zilles, Joel S. Emer* and Gurindar S. Sohi

Computer Sciences Department, University of Wisconsin - Madison
1210 West Dayton Street, Madison, WI 53706-1685, USA
{zilles, sohi}@cs.wisc.edu

*Alpha Development Group
Compaq Computer Corporation
emer@vssad.hlo.dec.com

Abstract

Common hardware exceptions, when implemented by trapping, unnecessarily serialize program execution in dynamically scheduled superscalar processors. To avoid the consequences of trapping the main program thread, multithreaded CPUs can exploit control and data independence by executing the exception handler in a separate hardware context. The main thread doesn't squash instructions after the excepting instruction, conserving fetch bandwidth and allowing execution of instructions independent of the exception. This leads to earlier branch resolution in the post exception code and additional memory latency tolerance. As a proof of concept, using threads to handle software TLB misses is shown to provide performance approaching that of an aggressive hardware TLB miss handler.

1 Introduction

Exception handling is a mechanism to flexibly, and with low implementation cost, handle "exceptional" events in a way that doesn't impact the execution of the common case. This is performed by inserting a short software routine into the dynamic instruction stream at the site of the exception. This exception handler resolves the excepting event so that the execution of the application can continue.

Current sequential execution paradigms provide no mechanism to insert the exception handler between instructions which are already in the process of execution. Typically, in-flight instructions younger than the excepting instruction are squashed and refetched following the execution of the exception handler (Figure 1.a). Like a branch mispredict, this significantly impacts performance in the locus of the exception; for a number of cycles after the exception is detected, fewer instructions are available for execution. Since many of the squashed instructions were control and data independent of the exception handler. This unnecessarily penalizes the execution.

To avoid squashing we require a mechanism which provides the appearance of sequential execution, namely correct dataflow and retirement order, despite the fact that the exception handler is fetched when the exception is detected. Correct dataflow implies observing true register dependences, but because little data passes between an application and an exception handler, often only values associated with the excepting instruction, a general purpose out-of-order register renaming mechanism is not required. The correct retirement order is different from the fetch order because the exception handler is retired before the excepting instruction. By allocating the handler to a separate thread, the desired retirement order can be enforced while maintaining FIFO resource management within a thread (Figure 1.b). Retirement is controlled so that the exception handler is retired in its entirety after all pre-exception instructions and before all post-exception instructions retire (Figure 1.c).

This work explores using separate threads in a multithreaded processor for exception handling to avoid squashing in-flight instructions. The exception thread does not have direct access to the application's registers, avoiding complex renamer hardware, and memory operations are executed speculatively, recovering if an ordering violation is detected. Although this mechanism is applicable to many classes of exceptions, in this paper we focus on software TLB miss handling. The multithreaded exception handling approach halves the cycles-per-instruction (CPI) attributed to software TLB miss handling, and with an optimization, which we call *quick-starting*, the performance discrepancy between software and hardware TLB miss handlers can be reduced by 80%. We expect similar benefits for other

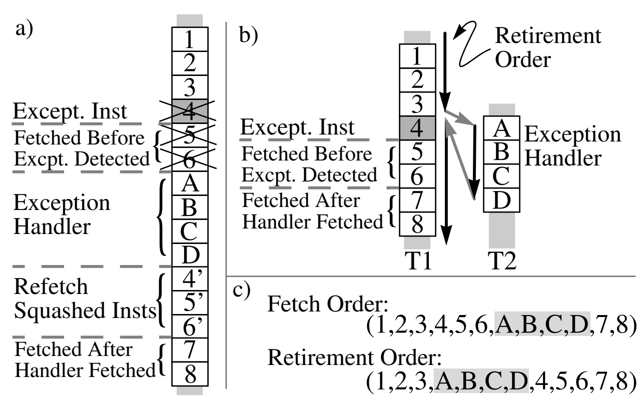

Figure 1. *Traditional vs. Multithreaded Exception Handling.* *Six instructions have been fetched when an exception is detected on the fourth. Traditionally (a), instructions 4-6 are squashed and must be refetched after the exception handler is fetched. With our multithreaded mechanism (b), a second thread fetches the exception handler (A-D), and then the main thread continues to fetch (7,8). The exception handler is retired before the excepting instruction. (c) This makes the global retirement order different than the fetch order, but each thread retires instructions in it's fetch order.*

classes of exceptions, which cannot be implemented in hardware state machines.

This paper is organized as follows: In Section 2 we provide more background on exceptions, focusing on software TLB miss handlers. Section 3 motivates this work by demonstrating how the performance impact of traditional software TLB miss handling increases with current trends in microprocessors. In Section 4, we describe the hardware requirements for TLB exceptions and walk through the exception process. In Section 5, we present performance results for a multithreaded software TLB miss handler along with our simulation methodology and model. In Section 6 we briefly describe how to generalize the mechanism for other types of exceptions. Finally, in Section 7, we discuss the related work and, in Section 8, conclude.

2 Background

Exceptions are events which are either impossible or too costly to handle through normal program execution. An illustrative example is arithmetic overflow; software could test whether an overflow occurred after every arithmetic operation and branch to fix-up code if necessary, but this would add substantial overhead to the execution since such overflows are uncommon.

Since a purely software solution is not appealing, and a purely hardware solution aren't cost effective nor provide the flexibility required by many exceptions, a hybrid hardware/software solution is generally used. The hardware is responsible for identifying the exceptional event, at which point it halts the executing program and transfers control to a software routine called the exception handler. This handler attempts to rectify the cause of the exception and determines if and when to return control to the user application.

Exceptions can be separated into two classes: *un-recoverable* and *recoverable*. Un-recoverable exceptions, where the system cannot restore the application to a meaningful state, are infrequent, at most once per application, so their performance is not a concern. In contrast, recoverable exceptions can be called repeatedly to perform "behind-the-scenes" work on behalf of the programmer and can affect system performance. They, in general, have a control independent nature; after the exception handler is executed they return to the site of the exception. This reconvergent behavior enables our multithreaded exception handling architecture. If the exception handler does not return to the excepting instruction we cannot avoid squashing and re-fetching. Some examples of recoverable exceptions are unaligned access, profiling, and instruction emulation. In this paper we study TLB miss handlers.

To provide the virtual memory abstraction without substantially sacrificing performance, modern microprocessors include a translation lookaside buffer (TLB). The TLB serves as a cache of recently used translations (from virtual addresses to physical addresses). When a virtual address that is not currently mapped by the TLB is accessed, the processor handles the TLB miss by fetching an entry from the page table.

TLB misses occur because the TLB cannot map the whole application's address space; in fact, many machines cannot even map their whole L2 cache. As memories sizes continue to grow at an exponential rate, we expect program data sets to grow proportionally. TLB size, on the other hand, is limited by processor cycle time, power dissipation, and silicon area in proximity to the memory datapath. Most architectures support large pages, which can increase the amount of memory mapped by the TLB, but large pages have proven to be difficult to use and can reduce the utilization of memory due to internal fragmentation. Secondary TLBs can scale more efficiently with data set size, but execution of future applications will likely continue to stress the virtual memory sub-system, maintaining TLB performance as an important component of overall system performance.

A number of architectures provide TLB miss handling through a dedicated, hardware finite-state-machine. This structure is capable of walking the page table and writing a new entry into the TLB. Instructions which miss in the TLB are stalled while the hardware page-walk takes place; no instructions need to be squashed, and, in machines which permit out-of-order execution, independent instructions can continue to execute. This TLB widget competes normal instruction execution for the cache ports, making the core somewhat more complex.

In contrast, one feature common to some RISC architectures (Alpha, MIPS, Sparc V9) is the software-managed TLB. Software-managed TLBs save hardware and provide flexibility to software on how page tables are organized and, in some cases, allow software to control replacement policies. In addition, they can be used to simplify implementations of software distributed shared memory (DSM), copy-on-write, and concurrent garbage collection. In current processors for these architectures, the pipeline is flushed at the memory instruction which missed in the TLB. The software TLB miss handler is fetched and executed, and then the application is restarted with the faulting instruction. This serializing nature of the traditional software TLB miss handling is not intrinsic to the nature of TLB fills, but merely an artifact of the implementation.

This paper presents multithreading as an alternative to both traditional mechanisms. Multithreading has been proposed as a technique for tolerating latency, typically memory latency [14]. Recently, microprocessors which support multithreading have begun shipping [16]. By time-multiplexing resources between multiple program "threads," high aggregate throughput can be attained despite chronic stalls, because each thread's stalls tend to be independent. In addition, simultaneous multithreading (SMT) [9,17,19], unlike coarse-grained multithreading, provides the flexibility to issue instructions from multiple threads in the same cycle. This tolerates the lack of parallelism in the individual threads, further increasing throughput.

3 Motivation

In this section, we demonstrate that the performance of traditional software TLB miss handling is increasing at a slower rate than program execution as a whole. With TLB miss handling becoming an increasingly large fraction of execution, alternative mechanisms for exception handling become appealing.

At the 1998 Microprocessor forum, Compaq presented a breakdown of the execution time of the transaction processing benchmark TPC-C for their current and future Alpha-based products [1]. The enhanced micro-architecture of the out-of-order 21264 spent the same amount of time on trap handling as the in-order 21164 (at the same frequency), but due to the 21264's increased exploitation of ILP in the rest of the application the percentage contribution for traps increases from about 8 percent to about 13 percent. Increasing the clock frequency (of the 21264) and integrating the L2 cache (the 21364) do not significantly change the percentage contribution from its 13 percent level. This evidence implies a relationship between the sophistication of the core and relative overhead of exception handling; in this section we present simulation results which explore this relationship in more detail.

Dynamically-scheduled superscalar is the paradigm of choice for current high-performance microprocessors. The processors seek to achieve high levels of instruction level parallelism (ILP) by speculating past unresolved branches and relaxing artificial constraints to issue instructions out of program order. Typically, these machines maintain a "window" of instructions from the predicted dynamic instruction stream, from which ready instructions are selected for execution in the functional units.

Achievable ILP is strongly dependent on useful window occupancy, the number of instructions from the correct path in the instruction window available for execution. The traditional mechanism for executing exception handlers reduces useful window occupancy by squashing all post-exception instructions. As machines continue their current trends of increasing superscalar width, window size, and pipeline length, the importance of keeping the instruction window full of "useful" instructions increases.

TLB fill latency is not a good metric for characterizing program execution time because it does not account for the extent that the execution of the TLB miss handler can be overlapped with other operations. To measure performance directly, we compare each simulation to one performed with a perfect TLB to identify the performance degradation due to TLB miss handling. Rather than dividing this penalty by the number of instructions executed, as would be done to compute the CPI contribution, we divide by the number of TLB misses. This "penalty cycles per TLB miss" metric allows comparison between benchmarks with widely different TLB miss rates. Details about our simulation model and benchmarks are available in Section 5.1 and Section 5.2, respectively.

Figure 2 shows the trends for increased pipeline length (3, 7, and 11 stages between fetch and execute, the minimum branch mispredict penalty) for an 8 issue machine. In correspondence to branch misprediction penalties, we'd expect longer pipelines to have proportionally higher TLB miss handling penalties, and we are not disappointed. The various benchmarks differ in their ability to tolerate the squashes, but the slope of the graph (i.e. its dependence on

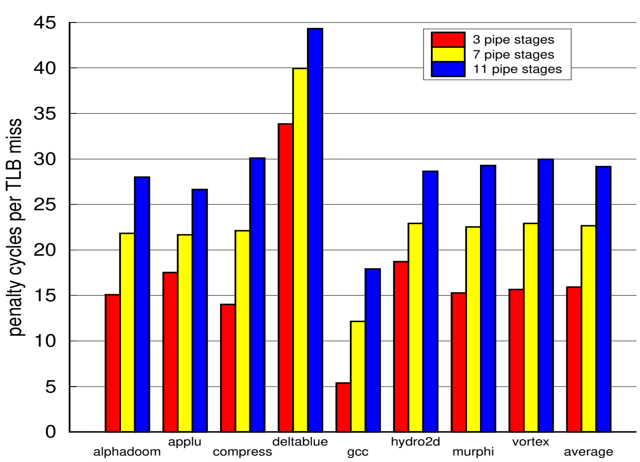

Figure 2. *Overhead of software TLB miss handling as a function of pipeline length.* *With an increasing number of stages between fetch and execute, overhead of traditional exception handling increases.*

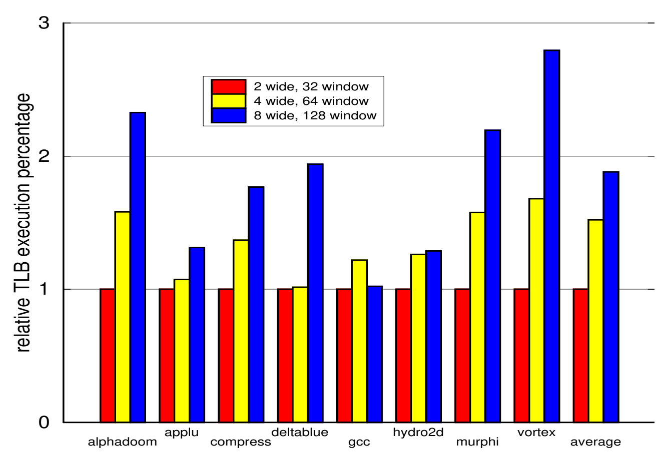

Figure 3. *Overhead of software TLB miss handling as a function of superscalar width.* *Wider machines spend larger percentage of their execution time on TLB miss handling because TLB miss handling does not benefit much from increased issue width.*

the pipeline length) is around 2 for most benchmarks. This roughly corresponds to the time to refill the pipe at the exception, and once again after the return from exception since our simulator does not have a return address stack (RAS) like mechanism for predicting the target of exception returns.

A similar experiment was performed to show the performance trends with respect to superscalar width. In general, as the machine width increases, the percentage of the time spent handling TLB misses increases. The wider machines are able to execute the miss handler slightly faster, but execution in the locus of a TLB miss does not benefit from wider issue nearly as much as the program as a whole. Figure 3 shows relative percentage of execution time spent handling TLB misses for 2, 4, and 8 wide machines with instruction windows of size 32, 64, and 128 respectively. The performance trend in *gcc* is a symptom of cache pollution from speculative memory accesses in the base case and is described in detail in Section 5.3.

These performance trends, which we expect to be representative for other types of exceptions as well, demonstrate a steadily increasing opportunity for alternate implementations for exception handling. Given that common exception handlers tend to be short, in the tens of instructions, purging the instruction window represents a significant portion of the exception penalty in future machines.

4 Description of the proposed hardware

In this section, we walk through the multithreaded exception handler's execution and discuss in detail the hardware requirements for this mechanism. This work is presented as an extension to a simultaneous multithreading (SMT) processor. For clarity, we focus on the execution of a software TLB miss handler.

4.1 Starting/Stopping Threads

When a TLB miss occurs, the faulting memory instruction (and any dependent instructions which have been scheduled) has to be restored to a state from which it can re-execute when the TLB fill has completed. This means returning the instruction to the instruction window and marking the instruction as not ready to execute. A similar recovery mechanism is required for machines which speculatively schedule instructions that are dependent on loads, before it is determined whether the load has hit in the cache [12]. Presumably, an extension to such a mechanism could support TLB misses as well.

To accomplish the fill, a thread context is selected and instructed to begin fetching the TLB miss handler. The privilege level for the TLB miss handler may be higher than the application code; we assume a processor which allows instructions from multiple privilege levels to co-exist in the pipeline. This thread could be one of the general purpose thread contexts in the SMT, or it could be a special-purpose, reduced-functionality thread context. Typically, the processor would select from idle threads, but lower priority threads could be preempted to run exception handlers on behalf of higher priority threads, sacrificing their throughput to benefit the higher priority thread.

The thread ID and the instruction identifier of the excepting instruction are stored by the allocated thread (Figure 4). This execution of the TLB miss handler proceeds in parallel with the execution of independent instructions from the application thread. When the TLB write is complete, the faulting instruction is made ready and scheduled normally.

Instructions are not retired in the order they are fetched; the exception handler is spliced into the retirement stream of the application (Figure 1). When the excepting instruction is the next to retire, retirement from the application thread is halted. This can be detected by comparing the sequence numbers of retiring instructions with those of excepting instructions which have spawned threads (Figure 4). At this point the exception thread can begin retirement. When it has completely retired (signified by the retirement of a RETURN FROM EXCEPTION) it resets its thread state to idle, signalling that retirement of the application thread can recommence. Also, events which cause squashes (for example control mispredicts) check exception sequence numbers to reclaim exception threads if the faulting instruction has been squashed.

4.2 Register and Memory Communication

Inserting an arbitrary piece of code into the dynamic instruction stream requires re-renaming the instructions after the insertion. Any post-insertion instruction which reads a logical register defined by the inserted code will have a stale physical register number for one of its source operands. This stale ID should be updated with the physical register number assigned to the instruction from the inserted code. This operation is equal in complexity to register renaming.

Fortunately, exception handlers are not arbitrary pieces of code and do not require a general-purpose re-renaming mechanism. Specifically, the TLB miss handler only reads the virtual address generated by the faulting memory instruction (which is provided from a privileged register read) and the page table (which is not accessible by the application) and only writes an entry into the TLB (which

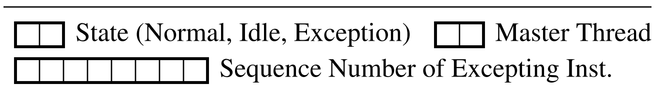

Figure 4. ***Additional Per-Thread Control State (in bits)****. When an idle thread is allocated to handle an exception, the ID ($\log_2$ (# of threads) bits) of the thread which caused the exception is put into* ***Master Thread****, and a "pointer" to the excepting instruction (~log2 (window size) bits) is stored.*

the application reads indirectly). The exception handler begins and ends with the same register state. The fact that no direct register communication takes place allows the hardware to be greatly simplified. The exception thread is provided with an independent set of registers which begin the exception handler containing undefined values. These registers are only used to hold temporary values.

Since there is no direct register communication between the TLB miss handler and the application thread, no ordering violations can occur in registers. On the other hand, because exception handlers can execute memory operations, we need to correctly enforce true dependencies between these memory operations and memory operations in the application thread. The TLB miss handler performs no stores and loads only from the page table. These loads will only alias with a store that is modifying or invalidating an entry of the page table. Since writing the page table already has special semantics, necessary to keep a coherent view of virtual memory in multiprocessors, ensuring proper ordering for the TLB miss handler's memory operations should be manageable.

4.3 Handling the Uncommon Case

Due to the concessions we made to simplify the hardware, the exception thread will not be able to handle all possible tasks that an exception handler might need to perform. If a TLB miss is discovered to be a page fault the process state needs be saved. To allow the application's registers to be read, we revert to the traditional exception handling mechanism.

This reversion process is complicated by the fact that it cannot be known *a priori* whether an exception will require the full functionality of the traditional mechanism or can be handled by an exception handler thread. Only by loading a page table entry can we detect that the desired page is not resident in memory, but at that point the handler has already been partially executed. Clearly, the mechanism must be able to recover from incorrectly spawning an exception handler thread for an exception which requires the traditional mechanism.

The handler needs a mechanism to communicate to the hardware that it requires traditional exception support. This could be accomplished completely in hardware (requiring the machine to recognize that the handler is trying to perform some action of which the multithreaded model is not capable) but is probably simpler to accomplish by providing software a mechanism which communicates this fact directly to the hardware. This *hard exception* mechanism could be in the form of an new instruction or a store to a special address, much like an inter-processor interrupt. This instruction should be present in the exception handler before any instructions which permanently affect the visible machine state.

When a *hard exception* instruction is encountered, the exception handler has been partially executed by the exception thread. The machine will either have to merge state from the application and exception threads together, or throw away the work in progress in the exception thread and re-fetch and re-execute the whole handler from the main thread. Since memory operands will have been prefetched into the cache, we expect the performance of a complete re-execution should not be substantially worse, and re-execution is likely to be significantly simpler to implement. The thread ID and instruction number stored by the exception thread (Figure 4) can be used to trigger a squash. To avoid repeating branch mispredictions in the re-execution, the branch outcomes could be buffered and used instead of a predictor for the previously executed region.

Some operating systems may choose to not implement spawnable exception handlers for all exceptions. For these exceptions, attempting to spawn the handler will only add latency to the exception handling; this can be avoided easily. The OS could explicitly inform the hardware which exceptions should be spawned by setting bits in a control register. Alternatively, the hardware could learn which exceptions should be spawned by tracking the use of the *hard exception* instruction. A small predictor, perhaps 2-4 bits for each of 16 or so exception types, could detect which exceptions were implemented with spawning in mind. In addition, it might be able to adapt to dynamic behavior, like clustering of page faults.

4.4 Resource Allocation

Once the handler thread has been allocated, it has to compete for processor resources. Previous work on SMT has shown that proper allocation of fetch bandwidth is critical for high throughput [17]. The handler thread should be given fetch priority over the main thread since the instructions in the handler thread will need to retire before any instructions after the exception[1]. Prioritization in the presence of multiple application threads is more complicated. Given a scheme like ICOUNT [17], which gives priority to threads with less in-flight instructions, it is likely that all active threads have approximately equal numbers of instructions in-flight. Statistically the excepting instruction should be at least as close to retirement as the most recently fetched instructions for other threads, implying that the exception handler, which should be retired before the excepting instruction, should be given fetch priority. This policy is naturally implemented in ICOUNT because when the handler thread is started it will have no in-flight instructions.

To avoid wasting fetch bandwidth, the handler thread should stop fetching once the complete exception handler

1. Although the exception handler can be launched control-speculatively, and may be squashed due to a branch mispredict, it is no more speculative than the post-exception instructions.

has been fetched. The common-case software TLB miss handler is typically in the tens of instructions long. By the time the RETURN FROM EXCEPTION instruction is decoded, signalling the end of the exception, multiple cycles worth of instructions past the end of the handler could have been fetched. To avoid the performance impact of these lost fetch cycles (approximately 0.5 cycles/miss), the machine could predict, using the previous handler execution, the number of fetch cycles an exception handler requires and prevent additional fetching until the initial instructions have been decoded.

Out-of-order fetch provides the opportunity for deadlock unless instruction window resources are properly managed. Deadlock occurs if the window is full of post-exception instructions. Instructions from the handler thread, which must be retired before the instructions in the window, will never complete because they can not be inserted into the instruction window and executed. Even for cases when deadlock doesn't occur, performance will suffer if sufficient window resources are not available.

A mechanism is required to restrict the original thread from monopolizing window resources and reclaim them when necessary. Since other application threads are not dependent on the handler thread, they will continue to retire instructions (and hence reclaim window resources) regardless of the condition of the handler thread. Other application threads are ignored for instruction window management purposes. In our implementation, when an exception occurs, a "reservation" is made for the window resources required to hold the handler (using the prediction of handler length mentioned above). The main thread is prevented from inserting additional instructions into the instruction window if no unreserved slots are available. In addition, to avoid deadlock, if the handler thread ever has instructions which are ready to be put in the window, instructions from the tail of the main thread are squashed to make room (unless such a squash would kill the excepting instruction, in which case the exception handler is stalled). Such a squash is an extremely rare occurrence in our simulations.

4.5 Multiple Exceptions

Multiple exceptions can occur in the same window of instructions; to avoid unnecessary serialization these should be handled in parallel when possible. Our hardware model provides support for renaming and speculative execution for privileged architecture state. This allows the traditional mechanism to handle multiple exception handlers in parallel assuming their control flow is predictable, but it cannot dispatch them in parallel because the second excepting instruction will be squashed when the first handler is fetched. In contrast, the multithreaded solution does not need to squash the second excepting instruction, allowing both exception handlers to be launched immediately. In addition, the multithreaded solution can gracefully handle launching exception handlers out-of-order.

There are two implementation options to handle the case when more exceptions occur than idle thread contexts are available: 1) stall exceptions until threads are available or 2) handle the additional exceptions traditionally by squashing and re-fetching. Stalling exceptions introduces another deadlock case (when exceptions are detected out-of-order and the oldest is not allocated to a thread) to be avoided. This, coupled with the fact that the traditional exception handling mechanism is already required, leads us to advocate using the traditional scheme.

One case, particular to TLB miss handlers, is detecting TLB misses to the same page out-of-order, which occurs 1-2% of the time. To maintain correct retirement semantics, the handler should be retired before the first offending instruction and only affect the state of later instructions. Traditionally, the handler is squashed and re-fetched at the correct instruction boundary. Since the correct handler is already in-flight, the unnecessary delay of re-fetching can be avoided. Our proposed multithreaded hardware detects this situation and re-links the exception thread with earlier excepting instruction, by updating the sequence number of the excepting instruction (Figure 4). We believe this can be implemented with minimal additional complexity. Whether or not this relinking is supported, a mechanism for buffering secondary TLB misses will be required due to their prevalence.

5 Experimental Results

To demonstrate the performance benefit of multithreaded exception handling we performed a series of experiments using software TLB miss handling as an example. Since there is no benefit to spawning exception threads for instruction TLB misses, only data TLB misses are modeled.

5.1 Simulation Infrastructure

This research was performed using a simultaneous multi-threaded simulator evolved from the Alpha architecture version of the SimpleScalar Toolkit [2], version 3.0. This execution-driven simulator supports precise interrupts/exceptions which allows us to trap at TLB misses and run the TLB miss handler. The simulator supports enough of the 21164 privileged architecture [4] to run the common case of the data TLB miss handler from the 21164 privileged architecture library (PAL) code. The base simulated machine configuration is described in Table 1.

The simulator has an abstract front-end which provides the benefits of a trace cache without a specific mechanism. It is capable of supplying instructions from multiple non-contiguous basic blocks in the same cycle and the number of taken branches per cycle is not limited. To simplify

Core	Dynamically-scheduled simultaneous multithreading with 2 or 4 threads. All threads share a single fetch unit, branch predictor, decoder, centralized instruction window (with 128 entries), scheduler, memory systems, and pool of functional units. Fetch, decode, and execution bandwidth are equal, nominally 8. The fetch chooser policy is described in Section 4.4. Instructions are scheduled oldest fetched first.
Branch Prediction	YAGS[7] with 2^{14} entry table, 2^{12} exceptions with 6 bit tags, with perfect branch target prediction. Indirect branches predicted by an cascaded indirect predictor [6] with 2^8 entry table, with 2^{10} exceptions. Returns are predicted by a 64 entry checkpointing return address stack (RAS) [10].
Pipelining	3 cycles for Fetch, 1 cycle Decode, 1 cycle Schedule, 2 cycle Register Read for nominal 7 stages between Fetch and Execute.
Functional Units (Latency)	8 integer ALUs (1), 3 integer mult/div (3/12), 3 Float Add/Mult (2/4), 1 Float Div/SQRT (12/26), 3 Load/Store ports(3/2) for an 8 way machine. All functional units are fully pipelined.
Memory System	64 KB, 2 way set associative (32 B lines) L1 instruction cache, 64 KB, 2 way set associative (32 B lines) L1 data cache, up to 64 outstanding (primary + secondary) misses, L1/L2 bus is 16B wide giving a 2 cycle occupancy per block, 1 MB (64 B lines) 4 way set associative fully-pipelined unified L2 cache with a 6 cycle latency (best load-use latency is 12 cycles), L2/memory bus occupied for 11 cycles during transfer, 80 cycle memory latency (best load-use latency is 104 cycles)
Translation	Perfect ITLB, 64 entry DTLB. PAL instructions can co-exist in pipeline with user-mode instructions, TLB misses are handled speculatively, and TLB miss registers are renamed to allow multiple in-flight misses simultaneously. Assume common case (no page faults or double TLB misses), enabling perfect prediction of handler length.

Table 1. ***Base simulated machine configuration***

simulation, instructions are scheduled in the same cycle they are executed, which in effect provides perfect cache hit/miss prediction. To account for the delay required for a register file read, instructions are prevented from scheduling until a number of cycles after they are put in the window. Limited execution resources and bandwidth are modeled, but writeback resources are not. Instructions maintain entries in the instruction window until retirement and must retire in order, but retirement bandwidth is not limited. A multi-level cache hierarchy and request/transfer bandwidths between levels of the hierarchy are modeled. The page table entries are treated like any other data and compete for space in the cache as such.

The simulated machine includes a 64 entry data TLB, smaller than contemporary machines, to account for the moderately small data sets of the benchmarks. Since all results are presented in terms of cycle penalty per TLB miss, rather than absolute speedup, results are not significantly affected by TLB size. Using a smaller TLB increases the number of misses per simulated instruction. Three TLB miss handler mechanisms are studied:

- The traditional software TLB handler squashes all instructions from the TLB miss on, fetches the handler and then resumes the application code. Instructions are free to use translations speculatively, but the translation is only permanently entered into the TLB at retirement of the exception handler.
- The multithreaded TLB miss handler executes the handler code in a separate thread, when available; otherwise it reverts to the traditional mechanism.
- Lastly, for comparison, a hardware TLB miss handler is studied. The hardware scheme does not require instructions to be fetched, but requires memory system bandwidth, and its load from the page table must be scheduled like other loads. The finite state machine can handle multiple misses in parallel and speculatively fills the TLB if the faulting instruction hasn't been squashed by the time the translation has been computed.

5.2 Benchmarks

Five benchmarks, those with non-trivial data TLB behavior, were selected from Spec95. Three additional benchmarks from various sources (X Windows, verification, object-oriented [5]) are included for additional breadth. All benchmarks were run for 100 million instructions. To avoid the initialization phase of the programs, the simulations were started from checkpoints partway into the execution. The benchmarks are listed in Table 2.

5.3 Analysis

Figure 5 shows the relative performance for four different exception architectures across the benchmark suite. The traditional software TLB miss handler has an average TLB miss penalty (run time difference compared to a perfect TLB, divided by the number of TLB fills, as described in Section 3) of 22.7 cycles per miss. The hardware TLB miss handler usually has the best performance (the only exception is *gcc* which is described in the next paragraph) with a TLB miss penalty of 7.3 cycles, around a third of the traditional software handler. The multithreaded(1) solution, which has one idle thread available for exception handling is a significant improvement over the traditional mechanism with an average penalty of about 11.7 cycles per miss, or just over half of the traditional miss penalty. Additional

Name	Data Set	TLB misses	description
alphadoom (*adm*)	-playdemo rockin	11,000	X-windows first-person shooter game Doom, from Id Software.
applu (*apl*)	test input	16,000	parabolic/elliptical partial differential equation solver (SpecFP 95)
compress (*cmp*)	100000 q 2131	230,000	text compression using adaptive Lempel-Ziv coding (SpecInt 95)
deltablue (*dbl*)	5000	16,000	object-oriented incremental dataflow constraint solver (C++)
gcc (*gcc*)	jump.s	14,000	GNU optimizing C compiler, generating SPARC assembly (SpecInt 95)
hydro2d (*h2d*)	test input	23,000	astrophysics-hydrodynamical Navier Stokes solver (SpecFP 95)
murphi (*mph*)	adash.m	36,000	finite state space exploration tool for verification (C++)
vortex (*vor*)	persons.250	86,000	single-user object-oriented transactional database (SpecInt 95)

Table 2. ***Benchmark summary.*** *TLB misses records approximate number of TLB misses in runs of 100 million instructions.*

threads provide only modest benefit; the multithreaded(3) experiments, which have 3 idle threads, reduce the average the miss penalty to about 11.0 cycles.

The multithreaded mechanism has better performance than the hardware mechanism on the benchmark *gcc* because the hardware mechanism speculatively updates the TLB, and *gcc* suffers from many TLB misses on mis-speculated paths. The speculative loads that cause these TLB misses cause cache pollution in the perfect-TLB case (the TLB filters these speculative accesses reducing cache pollution) giving the perception that the TLB penalty for *gcc* is lower than other benchmarks.

Although the multithreaded mechanism regains much of the performance lost to traditional software miss handling, there is still a discrepancy between its performance and that of the hardware state machine. The multithreaded mechanism has a number of overheads not present in a hardware mechanism: latency for fetching and decoding instructions, fetch and decode bandwidth, execution bandwidth, and instruction window space. To quantify these overheads we performed a series of limit-study style experiments, where we eliminated each of these overheads in turn and analyzed their effect on performance. These limit studies were performed with 3 idle threads to maximize performance.

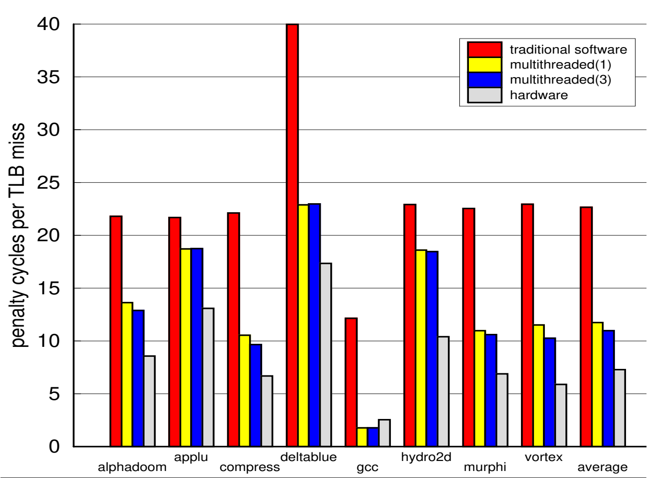

Figure 5. ***Relative TLB miss performance of traditional, multithreaded and hardware handlers.***

Table 3 shows the average results of the these experiments, comparing them with averages for traditional software, multithreaded software, and hardware mechanisms. Instantaneous fetch is the only optimization which significantly affects performance, reducing the miss penalty by 2.5 cycles. In the next section we propose a hardware optimization to reduce the fetch/decode latency of the software TLB miss handler.

5.4 Quick Start

Since fetch and decode latency is the major contributor preventing equivalent performance to the hardware TLB, we explored a possible optimization which reduces the latency incurred before the exception handler begins execution. Specifically we predict the next exception to occur, prefetch the exception code, and store it in the fetch buffer.

At fetch time, it can be difficult to predict whether there will be room for instructions in the instruction window given that it is unknown when instructions will retire. Our microarchitecture includes fetch buffers, which serve as a holding place for instructions which have been fetched but not decoded because the instruction window is full. In our SMT processor, these resources are supplied on a per thread basis. When a thread is idle, so is its buffer. These idle fetch

Configuration	Average Penalty/Miss
Traditional Software	22.4
Multithreaded	11.0
Multi w/o execute bandwidth overhead	10.7
Multi w/o window overhead	10.5
Multi w/o fetch/decode bandwidth overhead	10.2
Multi w/ instant handler fetch/decode	8.5
Hardware TLB miss handler	7.1

Table 3. ***Average number of penalty cycles per miss for different configurations.*** *The "Multi" configurations are limit studies of the multithreaded mechanism with one of its overheads removed. The latency of fetch and decode is a major contributor to the performance discrepancy between multithreaded and the hardware mechanism.*

buffers can be used to hold the exception code that was prefetched before the exception occurred.

To prefetch the exception handler, we have to predict which exception is likely to occur next. Since it is likely that a particular application is dominated by a small number of exception types, a simple history-based predictor is likely to perform well. Since our experiments only modelled data TLB misses, prediction of the next exception type was perfect and thus optimistic.

Figure 6 shows that the quick start mechanism does produce a sizable performance improvement, on average 1.7 cycles per miss. This improvement falls short of the instant fetch/decode limit study from Section 5.3, as the quick start mechanism cannot avoid the latency for decoding the exception handler, and the instructions have not always been prefetched.

Speedup is a function of the number of penalty cycles per miss, the TLB miss rate and the base IPC. Although we don't feel that these benchmarks, with their relatively small data sets, have TLB miss rates that are representative of important workloads, for completeness we have included Table 4 with speedups, TLB miss rates and base IPC.

5.5 Multiple Application Threads

Since an SMT processor will often be running multiple application threads, it is important to investigate the benefit of our technique in that environment. We performed experiments with 3 application threads (arbitrary combinations of our 8 benchmarks) and one idle thread. Figure 7 shows these results. The benefits of our technique here are more modest but are not unsubstantial, reducing the average TLB miss penalty by 25% (30% with quick start).

One thread proved to be sufficient for supporting 3 of these benchmark applications. The exception thread was active between 5 and 40 percent of the time, averaging about 20% activity.

There are many factors which affect these results. SMT processors are more tolerant of the delays caused by TLB misses because the other threads can continue to execute normally. This leads to a reduction of the overall penalty, reducing the opportunity for our optimization. However, because SMT tend to have higher throughput in general, the lost fetch and decode bandwidth due to unnecessary squashes becomes more harmful, hence our technique shows benefit. Similarly, the hardware TLB miss handler has an advantage over the software techniques because it doesn't allocate precious fetch and decode bandwidth to the exception handler itself.

6 Generalized Mechanism

In Section 4, we focused on the mechanisms necessary for TLB miss exceptions. Other exceptions, like unaligned access or floating-point exceptions, can't easily be implemented without some access to registers. Up to this point we've relied on the traditional exception mechanism for general purpose reading and writing of the register file, but the multithreaded mechanism could be extended to provide read access to the register file.

Since all threads in an SMT processor share a central pool of physical registers, the difficulty of providing cross thread register access is not in adding extra datapath, but rather finding the correct physical register number. The exception handler thread could be started with a copy of the application thread's register rename map as it existed immediately preceding the excepting instruction (mechanisms for copying register maps are proposed in [18]). The SMT will correctly handle these read-only registers naturally: the normal

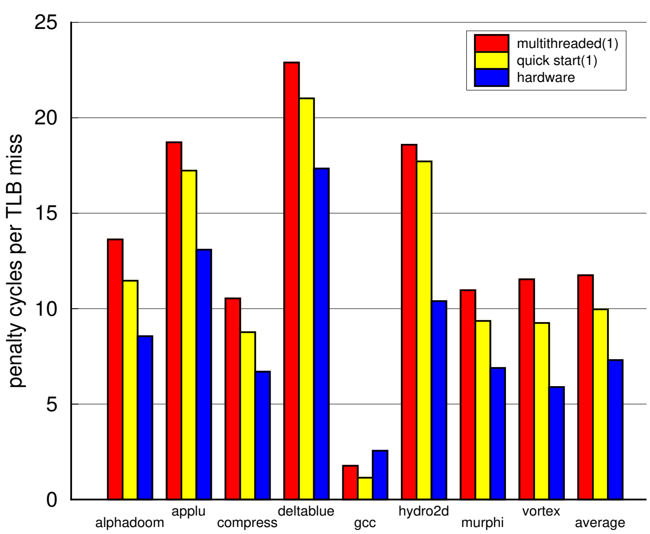

Figure 6. ***Performance of the "quick-starting" multithreaded implementation.***

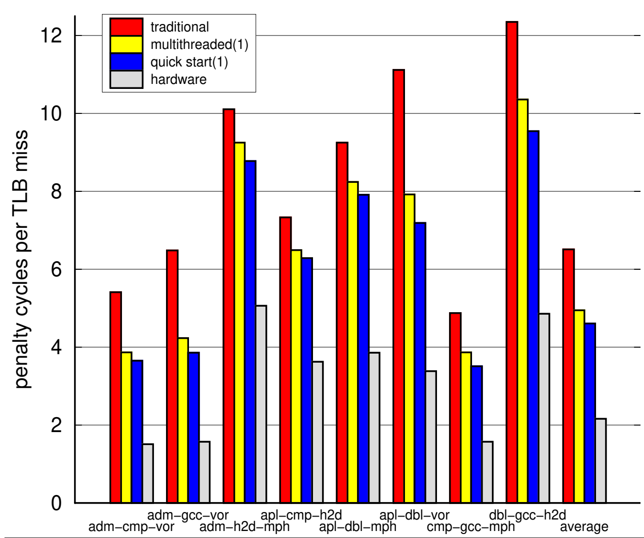

Figure 7. ***Average TLB miss penalties with 3 applications running on the SMT.***

scheduling mechanism handles the cross-thread dataflow, the registers will remain valid for the life of the exception handler since they can't be reclaimed until instructions after the handler retire, and the normal renaming mechanism will not allow the original thread's registers to be polluted. The difficulty with this solution is that the rename map might no longer exist when the exception is detected and may be expensive to regenerate. Although a mechanism to roll back the renaming maps to an arbitrary instruction boundary is necessary for traditional exception handling, utilizing this hardware to recover renaming maps for running threads might add unacceptable complexity.

Some exception handlers, including those for unaligned access and emulated instructions (those implemented in software), only need to read and write the source and destination registers of the faulting instruction. Simpler hardware could be built which would provide access to only those registers which were involved in the excepting instruction. In any machine in order to execute, an instruction needs to know the IDs of its source and destination physical registers; when an exception occurs we keep track of those register identifiers. In this way we can provide read access to the excepting instruction's source registers without the need to reconstruct the whole register map.

Write access to the instruction's destination can be similarly provided. When the exception is detected, the faulting instruction's destination register is recorded and it and all dependent instructions are returned to the instruction window. We can provide the exception handler a mechanism to write directly to this physical register. Upon this write, the excepting instruction is converted to a nop (to make sure that the register is not re-written) and any consumers of that register that are in the instruction window are marked ready and scheduled normally.

Other types of exceptions also need a more general memory ordering solution. Typically, dynamically scheduled processors include a mechanism to support out-of-order memory operations within a thread, but this needs to be extended to handle RAW violations between the exception thread and the application thread. This inter-process memory ordering support is already present in machines which enforce sequential consistency in the presence of speculative memory operations [20].

7 Related Work

A wealth of research has been done on multithreading and simultaneous multithreading, in particular, for increasing throughput of multi-programmed workloads and multi-threaded applications. Recently, Chappell et. al. [3] and Song and Dubois [15] have investigated mechanisms which allow subordinate threads to assist the execution of an application thread. This work differs from the previous work in three ways: 1) the exception threads are full-fledged SMT threads which are idle, rather than specialized threads with reduced register files [3] or a register file which is partially shared with the application thread [15], 2) instructions are fetched from the instruction cache; no micro-RAM has to be managed [3], and 3) the threads are synchronous; instructions executed by the subordinate thread are inserted into the application thread's retirement stream, and all synchronization between threads is implicit.

Significant work has been done in TLB design to reduce the frequency of TLB misses. Multithreaded TLB miss handling does not reduce the number of TLB misses, but instead reduces the performance impact of each TLB miss.

Previously, Henry explored mechanisms which accelerated interrupts and exceptions in a superscalar processor using the traditional mechanism [8], including tagging all in-flight instructions with a kernel/user bit rather than using a global kernel/user bit to avoid flushing the pipe at the transition. This mechanism is assumed in our implementation.

Concurrently with this work, Keckler *et. al.* performed a study on the performance impact of using separate threads for exception and interrupt handling for the M-Machine [11]. Because the M-Machine is an in-order machine, the work relies on the “instruction slack” between the excepting instruction and the first instruction which requires its result to overlap the handler with the faulting thread.

Our proposed mechanism exploits the control independence present in exception handler execution. Micro-architectures with general mechanisms for exploiting control independence [13] should be able to likewise exploit this aspect of exception handlers.

8 Conclusion

This paper presents a new exception architecture which uses idle threads in a multithreaded processor to execute exception handlers. The exception handler is executed in a separate thread, but instructions are forced to retire in the correct order maintaining the appearance of sequential execution. Squashing and re-fetching instructions after the faulting instruction is avoided, and, with dynamic scheduling, independent operations can continue to execute in parallel with the exception handler.

This execution model only applies to exceptions which return to the excepting instruction and limits access to register values from the main thread. Despite these limitations, this architecture seems promising for accelerating the execution of the classes of exceptions which are frequently executed.

The performance of this mechanism applied to software TLB miss handling is investigated. The overhead of traditional exception handling is rapidly increasing given the current trends in microprocessors. With the multithreaded

Name	base IPC	TLB misses	Perfect	H/W	Multi(1)	Multi(3)	Quick(1)	Quick(3)
alphadoom	4.3	11,000	1.0%	0.6%	0.4%	0.4%	0.5%	0.5%
applu	2.6	16,000	0.9%	0.4%	0.1%	0.1%	0.2%	0.2%
compress	2.6	230,000	12.9%	9.0%	6.8%	7.3%	7.8%	8.4%
deltablue	2.2	16,000	1.4%	0.8%	0.6%	0.6%	0.7%	0.7%
gcc	2.8	14,000	0.5%	0.4%	0.4%	0.4%	0.4%	0.4%
hydro2d	1.3	23,000	0.7%	0.4%	0.1%	0.1%	0.2%	0.2%
murphi	3.9	36,000	3.2%	2.2%	1.6%	1.7%	1.8%	1.9%
vortex	4.9	86,000	9.6%	7.1%	4.8%	5.3%	5.7%	6.3%

Table 4. *Table of speedups (over traditional software), TLB miss rates and base IPC for 100 million inst. runs of the benchmarks.*

mechanism the TLB miss penalty can be reduced by a factor of two. With a small optimization, speculatively fetching the exception handler and storing it in an idle thread's fetch buffer, the penalty can be further reduced, rivaling the performance of hardware TLB miss handling. When multiple applications are being executed the benefit is reduced to a 25% reduction of average TLB miss penalties.

Acknowledgements

We thank Amir Roth, Milo Martin and the anonymous reviewers for their comments and valuable suggestions on earlier drafts of this paper and Rebecca Stamm and George Chrysos for providing insight into SMT. This work is supported in part by National Science Foundation Grant MIP-9505853, and an equipment donation from Intel Corp. Craig Zilles was supported by an NSF Graduate Fellowship.

References

[1] P. Bannon. Alpha EV7: A Scalable Single-chip SMP. MicroProcessor Forum, October 1998.

[2] D. C. Burger, T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[3] R. Chappell, J. Stark, S. Kim, S. Reinhardt, Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In Proc. of the 26th Annual International Symposium on Computer Architecture, May 1999.

[4] Compaq Corporation. 21164 Alpha Microprocessor Hardware Reference Manual. order number EC-QP99C-TE, http://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html, December 1998

[5] K. Driesen, U. Holzle. The Direct Cost of Virtual Function Calls in C++. In Proc. of OOPSLA '96, October 1996.

[6] K. Driesen, U. Holzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In Proc. of the 31st Annual International Symposium on Microarchitecture, December 1998.

[7] A. N. Eden, T. Mudge. The YAGS Branch Prediction Scheme. In Proc. of the 31st Annual International Symposium on Microarchitecture, December 1998.

[8] D. S. Henry. Adding Fast Interrupts to Superscalar Processors. Computation Structures Group Memo 366, Laboratory for Computer Science, M. I. T. December 1994.

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

[10] S. Jourdan, T. Hsing, J. Stark, Y. N. Patt. The Effects of Mispredicted-Path Execution on Branch Prediction Structures. International Journal of Parallel Programming, vol 25, num 5, 1997.

[11] S. W. Keckler, W. J. Dally, A. Chang, W. S. Lee, S. Chatterjee. Concurrent Event Handling Through Multithreading. IEEE Transactions on Computers, November 1999.

[12] R. E. Kessler. The Alpha 21264 Microprocessor. IEEE Micro, Vol. 19, No. 2, March/April 1999.

[13] E. Rotenberg, Q. Jacobsen, J. Smith. A Study of Control Independence in Superscalar Processors. In Proc. of the 5th International Symposium on High-Performance Computer Architecture, January 1999.

[14] B. J. Smith. A pipelined shared resource MIMD computer. In Proc. of the 1978 International Conference on Parallel Processing, 1978.

[15] Y. H. Song, M. Dubois. Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, October 1998.

[16] S. Storino, A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S.Levenstein, G. Ulhmann. A Commercial Multi-Threaded RISC Processor. In Proc. of the IEEE 1998 International Solid State Circuits Conference. February 1998.

[17] D. M. Tullsen, S. J. Eggers, J. S. Emer, H M. Levy, J. L. Lo, R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In Proc. of the 23rd Annual International Symposium on Computer Architecture, May 1996.

[18] S. Wallace, B. Calder, D. Tullsen. Threaded Multiple Path Execution. In Proc. of the 25th Annual International Symposium on Computer Architecture, June 1998.

[19] W. Yamamoto and M. Nemirovsky, Increasing Superscalar Performance Through Multistreaming. Parallel Architectures and Compilation Techniques, June 1995.

[20] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor", IEEE Micro, 16, 2, April 1996, 28-40.