Enhancing Software Reliability with Speculative Threads

Jeffrey Oplinger and Monica S. Lam

Computer Systems Laboratory
Stanford University
{jeffop, lam}@stanford.edu

This paper advocates the use of a monitor-and-recover programming paradigm to enhance the reliability of software, and proposes an architectural design that allows software and hardware to cooperate in making this paradigm more efficient and easier to program.

We propose that programmers write monitoring functions assuming simple sequential execution semantics. Our architecture speeds up the computation by executing the monitoring functions speculatively in parallel with the main computation. For recovery, programmers can define fine-grain transactions whose side effects, including all register modifications and memory writes, can either be committed or aborted under program control. Transactions are implemented efficiently by treating them as speculative threads.

Our experimental results suggest that monitored execution is more amenable to parallelization than regular program execution. Code monitoring is sped up by a factor of 1.6 by exploiting single-thread instruction-level parallelism, and by another factor of 1.6 using thread-level speculation. This results in an overall improvement of 2.5 times and a sustained 5.4 instructions-per-cycle performance. A monitored execution that used to be 2.5 times slower executes with a degradation of only 12% when compared to the performance on the baseline machine. We also show that the concept of fine-grain transactional programming is useful in catching buffer overrun errors through a number of real-life examples.

1. INTRODUCTION

Developments in semiconductor technology and computer architecture over the past years have led to tremendous increases in computing performance. Today's high-end microprocessors offer much more computing power than the vast majority of existing software applications can fully utilize. On the other hand, non-performance metrics such as security, availability, reliability, and usability have become much more important. Errors and vulnerabilities in software programs have caused tremendous losses of data and productivity in the workplace, catastrophic mission failures and much more. Researchers are beginning to look at how the abundance of computing power that is available can be leveraged to address these "quality-of-life" computing issues.

Despite much work in the past on code verification and error detection tools, it remains the case that only small amounts of critical software can be proven to be correct. As the old saying goes, "to err is human." We believe that program errors can never be fully eliminated from complex software. As a complement to code verification techniques, we advocate introducing additional code into programs to monitor whether they are behaving correctly and to recover from errors. Because code monitoring can add significant overhead to a program's execution time and error recovery code can be complex, we propose the use of computer architecture support to make this "monitor-and-recover" style of programming both more efficient and easier to write.

1.1 Contributions

In this paper, we identify two programming idioms that enhance software reliability and propose an architectural design to support these schemes.

Efficient fine-grain code monitoring. To monitor the execution of a program properly, it is often necessary to invoke monitoring functions at relatively fine granularity throughout the execution of a program. Monitoring may be performed at procedural or basic block boundaries or even for all memory operations. Unfortunately, monitoring can add many more operations to a program. The run-time overhead has curbed the use of monitoring in production code, and to some extent, even in the software development and debugging cycle.

Monitoring functions need to observe the main program's state, but unless anomalies are detected, they do not have any effect on the execution of the main program. Thus, these functions can often be run in parallel with the main program as well as with each other. To keep programming simple, we propose that programmers simply identify these monitoring codes and assume that they are executed sequentially like any other functions in the program. The hardware, however, will execute these monitoring functions in parallel with the original program speculatively.

The concept of thread-level speculation (TLS)[1, 10, 22, 25, 29] has been proposed previously to speed up the execution of sequential programs. As originally proposed, a sequential program is divided up into threads, which may or may not be dependent upon one another. The hardware executes the threads simultaneously, but monitors the read and write operations of each thread to detect any data dependences that are violated due to the parallel execution. If a hazard is detected, the hardware aborts the thread that is supposed to execute later in the original sequential execution, eliminating all its side effects. The advantage of this approach is that the hardware can follow multiple threads of control at the same time, allowing operations that are far apart to be executed simultaneously.

Because of the fine granularity of monitoring functions, we adopt an architecture most similar to the proposed DMT machine[1]. Our proposed machine builds on the concept of simultaneous multithreading (SMT)[33], where a dynamic scheduler allows threads to share a common pool of hardware resources. Data hazards between threads are handled by augmenting the functionality of the load-store queues. We use value prediction to minimize rollbacks due to data hazards.

Fine-grain transactional programming. Writing error recovery code is difficult because the code must reverse any side effects that are no longer desired. Borrowing the concept of transactions from databases, we propose that a program be structured as a series of *transactions*. Each transaction is a unit of computation whose side effects, which include all changes made to registers and memory, can be committed or discarded as a unit. Error recovery can be achieved by simply applying an end-to-end check of the integrity of the computation and rolling back the transaction to its initial state if necessary.

To support transactional programming, we introduce a number of instructions with which the software determines when to start a transaction and ultimately whether to commit or abort. The TLS machine executes the transaction as a speculative thread; the side effects are saved in the speculative buffers, which can then be committed or discarded. The size of these buffers limits the amount of side effects that can be allowed in a transaction, so we suggest using the processor data cache to buffer larger amounts of transactional state.

Empirical validation. We have evaluated our proposed architecture through a number of case studies. Our experiments with four different examples of execution monitoring show that the hardware reduces the overhead of monitoring overall by a factor of 2.5, and even provides a respectable factor of 1.6 speedup once single-thread instruction-level parallelism (ILP) benefits are factored out. Our machine consistently executes 5.4 instructions per cycle (IPC), which is significantly greater than typical results obtained when TLS is applied to ordinary sequential programs. We also show that the concept of fine-grain transactional programming is useful in catching buffer overrun exploits through a number of real-life examples.

Evolution of new programming paradigms. This research tries to look ahead to the future and proposes architectural concepts that promote new and more effective programming paradigms that are considered impractical on today's architectures. The more traditional approach to computer architecture research has been to tune our architectures according to the characteristics of existing applications. If we only optimize for today's applications, the machines will be ill-suited to new programming paradigms; on the other hand, without an efficient implementation, new paradigms are hard to get established. Our proposed research approach is an attempt to break this cyclic dependency; we hope this will lead to a better combination of programming paradigms and architectures that together improves our ability to create reliable software.

In the past, so much energy has been devoted to squeezing the last drops of performance out of existing applications that we are seeing diminishing returns from new architectural features that are proposed. Programs written in new paradigms have different characteristics, and thus provide opportunities for new architectural development. In particular, achieving significant performance gains by applying thread-level speculation to existing non-numeric programs has proven difficult. We show in this paper that the self-monitoring programs are more amenable to thread-level speculation.

1.2 Organization of the Paper

The rest of the paper is organized as follows. Section 2 discusses the concept of execution monitoring: its uses, associated programming tools, and the characteristics of the monitoring functions. Section 3 motivates the need for transactional programming and proposes some concrete high-level syntax for expressing transactions. Section 4 presents our proposed architecture. We discuss how the architecture supports procedural speculation and transactional programming, our value prediction scheme, and our cache design for storing the speculative state. Section 5 evaluates the machine and our primitives on some sample codes. We then discuss related work and conclude in Sections 6 and 7, respectively.

2. EXECUTION MONITORING

The concept of execution monitoring has been used in the past for a variety of purposes: architectural evaluations, off-line performance tuning, on-line optimization, program debugging tools, and dynamic program verification. Execution monitoring is so common that new programming languages as well as binary and bytecode rewrite tools have been developed to facilitate adding instrumentation to programs. Here, we advocate additional support for execution monitoring at the architectural level in order to make this paradigm more efficient.

2.1 Uses of Execution Monitoring

Practitioners often use profiling to understand the bottlenecks in their system and tune the performance of programs. Dynamic languages like Java have been shown to benefit greatly from dynamic optimizations. These optimizations rely on profile data gathered by instrumenting the program to identify the frequently executed code regions or frequently used parameters. Execution monitoring is also widely used in computer architecture research to determine how programs behave under different architectural models.

Apart from performance analysis and optimization, execution monitoring has also been used in a variety of ways to improve software reliability. Application-specific assertions are added to the code by programmers, and safe languages add checks to ensure, for example, that there are no null dereferences or out-of-bound array accesses during execution.

A number of execution monitoring tools have been developed to watch for common errors in programs. StackGuard is an example of a tool that stops buffer overrun attacks, which have been a major cause of recent software vulnerabilities[6]. StackGuard inserts checks into the code to ensure that the run-time stack has not been inappropriately tampered with. Execution monitoring tools that help find memory management errors are widely used. For example, Purify monitors memory accesses to detect memory leaks, duplicate frees and illegal access errors such as reading past the end of heap-allocated objects or from uninitialized memory[12].

While StackGuard and Purify are designed to find specific types of programming errors, DIDUCE is a more general tool that helps programmers find all kinds of application-specific bugs[11]. DIDUCE instruments Java bytecode to watch the data values accessed at various code points in the program. It creates a model of the correct behavior as the program runs, reports a potential error whenever it encounters a violation of the model extracted and then relaxes the model. By doing so, DIDUCE usually alerts the programmers of errors as they first appear, rather than waiting for the corrupted data to cause program exceptions or failure. This technique has been demonstrated to help programmers quickly diagnose bugs that result from algorithmic errors in handling corner cases, errors in expected inputs, and developers' misconceptions about the APIs of software components they use. DIDUCE also helps programmers find hidden errors, errors that can silently compromise the integrity of the result and cause even greater damage than obvious errors that crash a program.

Today, these execution monitoring tools are used mainly during the debugging phase of software development. In fact, there are also many advantage to including execution monitoring codes in production software. By detecting run time errors in the software, execution monitoring can prevent errors from silently compromising the integrity of the system and data. In addition, it may be possible for software to capture error information and send it back to the developers for analysis, alert the user to take precautions before a harder failure occurs, or even attempt to recover from errors. However, the large overhead introduced by execution monitoring has not only made execution monitoring infeasible in production software, but also limited its usage in software development.

2.2 Tools for Execution Monitoring

Extensive monitoring code is usually inserted automatically by a program. For example, a compiler might insert relevant safety checks at all appropriate code points. ATOM[27] and EEL[16] are examples of binary rewrite tools that help programmers instrument binaries. BCEL is an example of a Java bytecode rewrite tool[7]. These tools allow users to insert calls to user-supplied routines at specified code points, such as before or after basic blocks, procedure calls, or data accesses.

Support for execution monitoring is one of the motivations behind the development of aspect-oriented programming[15]. To enhance modularity of software, aspect-oriented programming allows programmers to create an *aspect* that groups together behavior that cuts across typical divisions of responsibility in a given program. For example, monitoring code to be inserted at the beginning and at the end of every function would be organized as an aspect. The aspect compiler would automatically weave these functions into the main computation. In this way, all the source code related to monitoring is written in one place, making it easy for the programmer to write and maintain the software.

2.3 Speeding up Monitoring Functions

Monitoring functions represent pure overhead if the program is correct. In the rare event that an error is caught, it is important that the monitoring function communicate the failed check to the original program. This is usually achieved via a raised exception, an error return code, or by just terminating the main program. Production software programs, especially server programs where availability is paramount, often check for unexpected inputs or conditions and then needs to recover without terminating the program.

To enhance the performance of these checks, we propose that the software, typically an instrumentation tool, convey to the architecture which functions are monitoring functions. The programmer writes code following the simple semantics that the monitoring functions and the normal computation are executing sequentially. In fact, on a TLS machine, each call of a monitoring function may spawn a new thread; the original thread executes the monitoring function and the new thread executes the code after the call *speculatively* in parallel. The TLS architecture ensures that the sequential semantics are preserved. The monitoring function only sees the program state at the beginning of the call and the subsequent changes that it itself makes; the state of the speculative computation is buffered and not observed. If a data or control hazard is detected, that is if the speculative thread uses data that is later produced by the monitoring function call, or if the speculative thread was not supposed to execute at all, it is rolled back and not allowed to commit.

Typically the monitoring function reads the state of the main computation and operates on its own data structures. It may communicate with the main function by returning an error code that indicates success or failure. These kinds of results can be very accurately predicted because the program is expected to be correct. Thus, the speculative thread should not typically need to be rolled back.

It is possible that multiple monitoring functions might be invoked in rapid succession. If they are pure functions that simply return a predictable result, then parallel execution is possible. However, some of these monitoring codes may update internal data structures that are then read by later monitor calls. The TLS hardware will guarantee the sequential semantics even if data hazards occur between the calls. We expect these dependences to be rather sparse, especially if the monitoring routines are spaced out with the main computation.

2.4 Summary

In our proposed model, programmers can create monitoring functions simply by assuming that these functions will be called and executed sequentially. A code instrumentation tool inserts calls to the monitoring functions and marks these routines as speculative. The machine then exploits the fact that monitoring functions are mostly independent from the main computation and uses thread-level speculation to overlap their execution. When the program is correct, the monitoring functions run faster because they are overlapped with the main computation; if the monitoring functions find an error, the hardware allows the error be processed as if the program had executed sequentially.

3. FINE-GRAIN TRANSACTIONAL PROGRAMMING

Once an error is detected, it is important for many programs to recover from the error and not just terminate the computation. For example, the software may be providing services that cannot be interrupted, or the program may have some volatile state that is difficult or impossible to reconstruct. Even for software that can easily be restarted, the overhead involved in restarting a process may represent a denial-of-service vulnerability.

Clearly, it is preferable that software check its inputs and not make any erroneous updates to its state. However, despite all precautions, programmers still make mistakes. It is often easier to introduce end-to-end checks that examine the integrity of a computational state, but that means the program must now recover from a compromised state.

Transactions are a powerful abstraction that can be used to address this difficulty. We envision that programs in the future be built as a composition of transactions. Each transaction is a unit of computation whose side effects, which include all changes to registers and memory, can be committed or discarded as a unit. If checking code detects a problem, the transaction is aborted and the pre-transaction state restored. This provides the programmer with an extremely simple approach to error recovery.

Transactions have been extensively used in database applications; these are rather heavy-weight transactions that typically involve writing to permanent storage. The concept of transactions has also been applied at the operating system level, where changes of virtual memories and even system calls can be “undone”[13, 17, 23].

Transactions used for the sake of error containment and recovery can be very fine-grained, on the order of tens and hundreds of instructions. As such, we must reduce the overheads involved in implementing these transactions. By keeping transaction state in the speculative buffers of a TLS machine, we can provide extremely low-overhead transaction support.

3.1 Examples of Transactional Programming

Let us motivate the use of end-to-end checks with the single most common source of security vulnerabilities: buffer overruns. Despite the large number of man-years devoted to solving this problem, additional security holes based on overruns continue to be discovered. Many vulnerable programs are written in C; because C is unsafe, these programs can often be manipulated with carefully crafted inputs to write past the intended buffers. Manual inspection has proven to be inadequate, because vulnerabilities have been found even in codes that have already been audited for security holes[6]. It is in general impossible to determine if a program may overrun its buffer statically. And dynamic techniques that insert array bound checks into C are too slow to use in practice[14].

The need for end-to-end checks has prompted the development of tools like StackGuard[6], which verifies that portions of the C run-time stack have not been improperly overwritten. Unfortunately, by the time a problem is detected, damage has already been done.

Using transactional programming, the programmer only has to identify routines that are potentially problematic, as opposed to finding the exact errors. For example, many servers have been found with security vulnerabilities in input parsing functions. One approach is to make the input handling routine a transaction and monitor its computation. If any errors are found, the entire transaction is aborted and that particular input can be skipped. The input handling routines can create necessary data structures as normal, knowing that all side effects will be wiped out cleanly, without any corruption of data structures, if the transaction is eventually aborted.

Transactional programming also makes it convenient to write programs that may need to "undo" portions of the computation. An example can be found in network protocol processing code. In the implementation of HTTP proxy caches, for example, there can be a considerable amount of effort dedicated to ensuring that HTTP inputs are reasonable. Unreasonable requests can result in crashes or the allocation of excessive or unauthorized resources. When an unreasonable request is detected, perhaps midway through processing, any partially constructed data structures that have been created must be cleaned up. By treating the processing of the request as a separate thread with its own private memory state, the model for such validation is simplified considerably. Input validation is considered a separate transaction that can be aborted, and when aborted the resources are reclaimed. This compares favorably to the conventional model of error handling "cleanup" code that is infrequently exercised or tested, and can contain subtle bugs or resource leaks.

3.2 Programming Constructs

To make the concept of transactional programming more concrete, we propose that the programmer writes transactions using the following syntax:

```
    TRY {
      ... the original code ..
      if (error-detected())
         ABORT;
    } CATCH {
      return an error code;
    }
    return ok;
 }
```

Although the `try..catch` syntax we have adopted above resembles that of traditionally exception handling constructs, the semantics is quite different. In conventional exception handling, the stack is unwound and the control flow returns to the `catch` block when an exception is signaled, but side effects made to all other data structures are not removed. The exception handler must be careful in restoring all the data structures to a clean state. In our case, the hardware automatically buffers up all the memory writes and discards them when an `abort` is issued, guaranteeing that the memory is restored to its state just before the `try` statement. It is this ability that allows the program to recover from attacks that overwrite data structures beyond those expected.

To support this operation, we propose three machine instructions:

TRY ⟨ADDR⟩. This instruction indicates the start of a transaction. In TLS terminology, the thread at the point becomes speculative and all the side effects are buffered. If the transaction is aborted, the speculative state is discarded and the flow of control branches to the given ⟨ADDR⟩ address.

ABORT. This instruction indicates that the transaction is to be aborted. The hardware discards the speculative state, and the program counter is set to the address specified by the last TRY instruction.

COMMIT. This instruction indicates that the transaction is to be committed. The machine commits the speculative state, and the execution continues as usual.

Compiling the high-level construct discussed above to machine instructions yields:

```
     TRY <L1>;
     ... the original code ..
     if (error-detected())
        ABORT;
     COMMIT;
     return ok;
 L1: return an error code;
```

3.3 Discussion

Different granularities of transactions require different implementation techniques. Architectural support is necessary to minimize the overhead of transactions of the finest granularity. Conversely, hardware support is unsuitable for implementing coarser-grain transactions. Statements nested under these architectural-supported transactions cannot execute any system calls unless operating system support is provided. If the code is run in a multi-threaded environment, threads can read shared state, but if a thread wishes to write into any shared state, then locks must be held until the speculative state is committed.

Ideally, we want to provide the programmer a system that automatically adapts and chooses a combination of architectural and operating system techniques to implement transactions of any granularity. One possibility is for the machine to raise an exception when the state of the transaction overflows the available hardware, and have the system switch to an operating-system based technique to support the coarse granularity.

4. THE ARCHITECTURE

There have been quite a number of architectures proposed for thread-level speculation. They all provide three basic functions:

- multiple simultaneous threads of control,
- buffering of speculative state so that side effects can be discarded and false dependences can be eliminated, and

- detection of true data dependence violations between threads and the ability to discard or commit the speculative state to maintain sequential semantics.

The various architectural proposals vary widely in the degree of coupling between the threads in the system. At one extreme, speculative thread parallelism has been proposed for large-scale shared memory multiprocessors[3, 9, 28]. More common proposals, such as those based upon chip multiprocessors, are fairly loosely coupled and typically provide inter-thread communication through the level-2 cache[5, 10, 29]. Some propose relatively novel architectures, such as the ring of processing elements found in the Multiscalar[25]. The most tightly-coupled implementations include the DMT machine[1], which adds speculative thread support to a simultaneous multithreaded (SMT)[33] version of a traditional superscalar processor. In the DMT, threads communicate through forwarded registers and an expanded load-store queue.

Our objective is to support the speculative execution of both execution monitoring code as well as fine-grained transactions. Both of these scenarios generate relatively fine-grained threads, which are likely to perform poorly on a loosely-coupled TLS machine. Thus, our proposed architecture is based on a tightly-coupled SMT machine, and whose design is similar to the DMT machine proposal.

TLS machine proposals also vary in the way they divide the computation up into speculative thread. Parallelizing loop iterations is the most common technique[10, 29, 32]. Some researchers have proposed speculating on procedure continuations, that is, the codes that immediately follow a procedure call[1, 19]. Other techniques include using a compiler analysis to divide the static program into threads[35], using fixed-interval chunks of instruction traces[22], or applying a predetermined heuristic to dynamically partition the program[4].

While our architecture can support all the above options, this paper focuses on the support of execution monitoring and transactions. Execution monitoring is a special case of speculative procedure continuations, the main difference being that software is responsible for selecting when speculation should be performed. Software can exploit higher-level program knowledge to pick the functions, in this case monitoring calls, whose speculation is more likely to succeed. The support of transactions, as discussed in Section 3, requires the addition of a few instructions so that at run time the software can control whether transactions are committed or aborted. These relatively small additions give the software the necessary hooks to make better use of the basic TLS hardware.

We now give an overview of the architecture, followed by the details of how the speculative threads are controlled, and the addition of value prediction to the scheme. Finally, we discuss how we support transactions.

4.1 Overview

Our architecture is an extension of a simultaneous multithreaded machine (SMT)[33], which in turn is based on a superscalar architecture. Figure 1 shows a block diagram of our machine. The main features of a superscalar machine include a standard 5-stage pipeline:

1. Fetch instructions from the instruction cache into the instruction queues.
2. Decode the instruction and rename registers.
3. Issue ready instructions and execute.
4. Write back register results.
5. Commit instructions in the original program order.

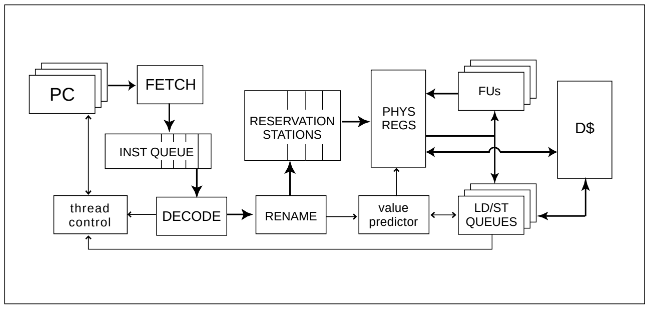


Figure 1: Machine Architecture

Simultaneous multi-threading extends the superscalar processor by allowing the machine to execute multiple threads on the same set of functional units, using primarily the same hardware scheduling mechanism as the superscalar. To store the contexts associated with the multiple threads, architectural registers (including program counters), the register renaming table, and the branch predictor's return-address stack are all duplicated for each thread. The fetch unit is modified to arbitrate instruction fetch between the threads, while branch misprediction recovery as well as instruction retirement are modified to operate on a per-thread basis. Having a common pool of physical registers and functional units, SMT allows the threads to share the hardware resources in a more fluid manner.

The following is a high-level description of all the features added to support thread-level speculation. More details are provided in Sections 4.2 through 4.4.

Thread control logic. This unit initiates threads, detects when threads stop, and either validates the speculation or restarts the speculative thread. It keeps track of the priorities of speculative threads, with higher priorities being given to those that would have occurred earlier had the code been executed sequentially.

Speculative memory state and data hazard detection. The load-store queue (LSQ) is replicated per thread to buffer the speculative memory state. This fine-grain support allows stores by a thread be observed by a later thread within two clocks. The queues are augmented with address comparators and snooping logic to detect memory data dependence violations between the threads. The original five-stage pipeline is augmented with a final commit stage that issues the buffered speculative stores to memory when the thread commits.

Additional architectural register sets per thread. We add a set of "input" registers per thread, representing the starting state of each thread. We refer to the per-thread architectural registers that are updated during thread execution (known as the "retirement" register file in a superscalar) as the "output" registers. We presume a register file organization very similar to the DMT's, where single-cycle copies of the registers from thread to thread are possible. Note that in TLS, the output registers are no longer known to be correct and final, as the speculative thread could potentially be restarted. A single "final" register set is updated when threads commit and are no longer speculative; no recovery is possible once final registers are updated.

Modified arbitration policies. As in typical SMT implementations[34], our machine can fetch from a maximum of 2 threads per cycle. With TLS, however, all threads are not created equal. Only the sequential thread is guaranteed to make forward progress, while all the other threads are speculative and may be rolled back. Thus our policy is to fetch from the two highest priority threads whose fetches are not blocked. We also prioritize the arbitration for shared resources each cycle to favor the higher-priority threads.

A value predictor. To improve the success of speculation, a value predictor is used to predict the results of procedure return values and load instructions that would otherwise cause thread squashes.

Buffered transaction state in the cache. To support coarser-grain transactions, each cache line is augmented with a single bit to mark whether the data is speculative.

4.2 Procedural Speculation

Procedural speculation is supported with a design very similar to that in the DMT architecture. The design supports nested procedural speculation as well as exception handling.

In procedural speculation, it is the code following the procedure that is being executed speculatively, since the called function executes first according to the original sequential semantics. A thread's current context is used to execute the called function, and a new thread context speculatively executes the computation after the call.

A speculative thread may encounter three kinds of potential hazards: memory data hazards, register data hazards, and control hazards. Memory data hazards are detected as a thread runs, whereas register data hazards are checked only when a thread is ready to commit. A thread is rolled back whenever a data hazard is detected; when that happens, all the lower-priority threads that have read from other speculative threads must also be rolled back. A control hazard occurs if the procedure call preceding a speculative thread generates an exception and never returns to the call site. In that case the speculative thread stays in the system until it is evicted from the machine.

The lowest-priority thread is evicted whenever a higher-priority thread wishes to spawn but there are no thread resources available. This ensures that resources are applied to the most important and least speculative threads. Useless threads wind up being evicted from the machine, but poor load balance or control mispredictions can cause useful threads to be evicted as well.

A thread executes until it reaches the end of the program, or it *meets* the speculative thread started on its behalf, or it gets evicted from the system. Its state is committed when it becomes the *head thread*, that is, the thread with the highest priority in the system. We describe each of the operations in more detail below.

Creating a new thread. To create a new thread, an empty thread context is selected if available. If all contexts are full, a new context is made available by discarding the lowest-priority thread. This requires invalidating all its state in the reservation stations, execution units, load-store queues, rename tables and fetch structures.

The "output" register state of the current thread is flash-copied in a single cycle into the "input" register set of the new thread. We also keep an additional set of "input" register renaming tables per thread, also flash copied from the current thread's mappings, so that cross-thread input register mappings are not lost if a thread mis-speculates and must be restarted. The new thread's program counter is set to the instruction after the call (the return address) and it begins fetching from there. In addition, the return address stack of the current thread is copied to the new thread. Our system allows only one new thread to be created per cycle, again favoring higher-priority threads, and there is no additional cost to starting a thread, apart from the pipeline startup delays associated with redirecting fetch.

Stopping a thread. During the execution of a thread, the thread control logic watches the fetch addresses of all the threads. If any thread T_i attempts to fetch from the same address as the start PC of the next speculative thread T_{i+1} in the priority list, instruction fetch for thread T_i is blocked and T_i waits to become highest priority. We refer to this as a thread *meet*. Note that a stopped thread may be restarted if data hazards arise before the thread can be committed.

Committing a thread. When a speculative thread becomes the head thread T_1, the thread is no longer speculative and the final commit stage becomes active. In that stage, any buffered speculative stores are issued to memory, up to the commit width of the machine and subject to the availability of store ports. T_1's speculatively written output registers are also committed to the final register file, again subject to the commit width. Note that T_1 can continue to fetch and execute while the thread is committing, as long as fetch has not been stopped by a meet with the next thread T_2. If T_1 finishes committing and has not met with T_2, it continues executing as the sequential thread and thus does not buffer any speculative state. The store addresses produced by T_1 will still be snooped by lower-priority threads to detect potential hazards.

Memory data hazards. A lower-priority thread must watch the writes of higher-priority threads to ensure that there are no data hazards. These hazards are detected in the load-store queues. All loads and stores of a speculative thread are buffered in program order in a single queue, and these queue entries are not released until the speculative thread commits. Store entries contain both the address and the data for the store. Load entries contain the address as well as a bit indicating whether the load was satisfied by an earlier store entry in the same queue.

When the address of a store in the LSQ resolves, it is checked against all loads with resolved addresses in the queues of lower-priority threads. The following two conditions are checked:

1. Do the addresses match or overlap?
2. Was the load satisfied within its own thread by a earlier store in the same queue?

If condition 1 is true and condition 2 is false, then there is in fact a data hazard, and the lower-priority thread (with the load) is restarted (using its saved "input" register state) so that the memory dependence will be observed. In addition, any other lower-priority threads that have read any speculative data from other threads are restarted as well, as they may have read corrupt data from the thread we are restarting. Note that we do not restart when both conditions 1 and 2 are met. This effectively implements "memory renaming" between the threads, and prevents some unnecessary restarts due to false dependences.

Register data hazards. Register validation takes place when the head thread T_1 has both finished the commit process and met with the second-priority thread in the list T_2. At this point, we know that any memory dependence violations would have been detected and handled, and that the final registers have been updated by T_1. We compare the final register state written by T_1 with the input register state of T_2. If any register values do not match, and have been read by T_2, then T_2 must be restarted with the final register values. If the register values match, T_1's context is freed, T_2 becomes the head thread, and then begins its commit process.

Performance optimizations. If a thread is restarted due to a data hazard, any threads it has created in the past are now unneeded *orphan* threads, as the restarted thread will create any desired child threads again as it reexecutes. Because of nested thread creation, however, there may be valid threads, representing speculation at outer procedure levels, that are lower in the priority list than these orphans. Because of that lower priority, they would be targeted for eviction sooner than the orphaned threads. To prevent this, the control logic for each thread watches the status of its parent thread. If a parent is restarted or deleted, the newly orphaned thread will stop execution as soon as access to the thread priority list is available. Transitively, any children of the orphaned threads will be deleted in later cycles.

4.3 Value Prediction

All new threads receive a starting copy of the spawning thread's register state at the time they are created as its predicted starting state. We refer to this primitive form of value prediction as "spawn prediction". For procedural speculation, this prediction tends to work very well, with the exception of the return value register. The speculative threads would have restored caller-saved registers before using them, and callee-saved registers would be restored by the callee before it returns if it changes their values. Thus, no data hazards are expected to be observed for any of the register values except for the return value. In addition, data hazards may exist through updates to memory (e.g. pointer arguments or global variables).

To improve the success of speculation, we have implemented value prediction for both procedure return values and load instructions that cause thread restarts. Entries in the predictor are indexed by the thread-starting address for return value predictions and the instruction address for load predictions. When a thread is initiated or a load is dispatched, the predictor is checked for an entry. If there is a hit, and the confidence values for the prediction entry are sufficiently high, the predicted value will be installed in the return value register for procedure threads, or in the destination register for loads. The value predictor used is the Hybrid Local/Global predictor described in[5], but augmented with a 2-bit saturating dependence confidence predictor.

If thread T_1 meets with thread T_2, and a return value prediction had been made for T_2, the standard register validation process (described in Section 4.2) is used to check the values and restart T_2 if necessary. In addition, the value predictor is accessed and updated with the correct return value from T_1.

Load value validation is accomplished by keeping in the load/store queue both the "latest" or most recent value written to the load address by an earlier thread, as well as the predicted value that used for execution. A predicted load will still issue to the LSQ and initialize the "latest" value with the most recent matching store in the queue, or with the value from memory if there is no match. As stores from higher-priority threads commit in sequential program order, the load/store queue will be checked for address matches with any predicted loads in future threads. Without value prediction, these matches would cause a thread restart, but with load value prediction, we simply update the "latest" value for the load and allow that thread to keep running. This has the effect of hiding "silent" stores that do not present a true data dependence.

When the thread is ready to commit its speculative state, the "latest" load values are now indeed correct, and can be compared with their respective predicted values. Any incorrect predictions will cause the thread to discard its state and restart.

Entries are introduced into the value predictor on demand when a thread is restarted either because of an incorrect return value register (after simple "spawn" value prediction has failed) or because of a memory dependence violation. The value predictor is updated as a thread commits its speculative state. The dependence confidence is incremented if a correct prediction did (or would have) prevented a violation and restart, or decremented if a prediction (even if correct) would not have been needed to prevent a violation.

We should note that the load value predictor was motivated more by traditional TLS speculation on normal sequential code than by monitor-and-recover applications. The instrumentation examples we present here do not make extensive use of value prediction for load instructions.

4.4 Speculative Transaction Implementation

For transactions, all the important decisions are made by the software; programs use the TRY⟨ADDR⟩ , COMMIT, ABORT instructions to dictate when the speculation starts, when to commit the speculative state, when to abort the speculation. Unlike the case of procedural speculation, the hardware does not have to detect data hazards nor initiate roll backs on its own.

Let us first consider the simple case when transactions cannot be nested. When a TRY⟨ADDR⟩ instruction is issued, the machine does not need to start another thread, it only needs to save the current state in case an abort instruction is issued. This is achieved on our hardware by flash-copying the current register state into its input register state. Register mappings are also flash-copied, and any outstanding register writebacks (resulting from instructions issued before the transaction began) must update both the current and the input registers. In addition, the address supplied in the Try statement is also stored as the target should an abort be issued. From this point on, any stores that are issued are part of the speculative state.

If the program executes an ABORT instruction, the speculative memory state is discarded, the registers are restored to the saved input register values, the program counter is set to the address the user specified with TRY, and the thread proceeds with normal non-speculative execution. If COMMIT is executed instead, the speculative memory state is committed, and the thread resumes normal execution.

While architecture-supported transactions are very efficient, they are limited by the amount of speculative state that can be stored in the hardware. The load/store queues, which hold the speculative state for procedural speculation, are expensive because of the logic needed to support data hazard detection. (In our simulated architecture, we assume that every thread has a load/store queue holding only 64 entries.) With traditional TLS, we can always discard a speculative thread if the speculative state storage is exhausted, but that is not the case with transactions.

It is thus highly desirable to increase the amount of speculative buffering to support coarser-grain transactions. We observe that data hazard detection is unnecessary for the implementation of transactions. All we need to do is to keep the speculative memory state from overwriting the sequential state. We can expand our speculative storage with a simple extension to the data cache.

All data written into the cache during speculative execution is labeled "speculative". If a transaction is commits, all the speculative bits in the cache are cleared. If a transaction is aborted, however, the speculative cache lines are invalidated. A cache line that is marked speculative cannot be written back to the memory hierarchy. If such a line must be evicted from the cache due to conflict or capacity issues, an exception is raised indicating that there are insufficient architectural resources to handle this transaction.

A single "speculative" bit per cache line is all the additional storage that is required for single-threaded transactional programs. To support transactions under multi-threaded execution, a bit per thread would be required. Similarly, nesting of transactions may be supported by introducing more bits per thread. We currently use the level one cache for this purpose; this same approach can be applied to all the levels of the cache. If very coarse-grain transactions are desired, this approach could potentially be adapted to the management of the virtual memory at the operating system level.

Parameter	SMT1	SMT2	SMT4*	SMT4
fetch width	4	8	16	16
issue width	4	8	16	16
commit width	4	8	16	16
res stations	128	256	512	512
lsq entries	64 p.t.	64 p.t.	64 p.t.	64 p.t.
int ALUs	4	8	16	16
FP ALUs	4	8	16	16
mem. ports	2	4	4	8
fetch ports	2 ports, priority scheduled			
branch predictor	8k/8k entry gshare / bimodal, 8k meta predictor			
value predictor	128kb AMA[5]			
L1 Dcache	32k, 4-way SA (32B lines), 1 cycle latency			
L1 Icache	32k, 2-way SA (32B lines), 1 cycle latency			
L2 cache	512k, 4-way SA (64B lines), 6-cycle latency			
memory	100 cycles latency			

Figure 2: Parameters of the Simulation

5. SIMULATION RESULTS

We have implemented a simulator of our architecture based on the SimpleScalar 3.0c out-of-order simulator for the Alpha instruction set[2]. We extended the simulator to support simultaneous multithreading and thread-level speculation as discussed in Section 4. System calls are executed by trapping through to the host operating system and cannot be executed speculatively. Thus speculative threads must be delayed until they become non-speculative if they attempt to make a system call. System calls are not allowed inside transactions.

Figure 2 shows the simulator parameters. Parameters labeled "p.t." represent per-thread values for non-shared resources, so an SMT2 processor with four threads would have 64 load-store queue entries per thread, or $64 \times 4 = 256$ entries across the whole machine. We simulate four sample SMT configurations, with the larger configurations typically having two or four times the overall base SMT1 resources. Memory interfaces are more limited. As discussed in Section 4.1, the fetch unit services a maximum of two threads per cycle. For the data cache, the SMT4* configuration still has four memory ports, but we also present results for the more unrealistic SMT4 machine with eight memory ports. Note that in the text, we use the shorthand SMT*m*/t*n* notation to refer to an SMT*m* machine running with *n* thread contexts available.

Benchmarks we tested were all compiled using gcc 2.91.66 with `-O2` optimization. The version of ATOM used was 2.17d. Complete instruction counts were obtained using a fast functional simulator, while performance statistics were gathered from simulations of the first 1.5B instructions of the program.

5.1 Monitoring Functions

Our first set of experiments evaluate the performance of applications after adding instrumentation with the help of an automated tool. We experimented with three execution monitoring tools. The first two, Pixie and Third Degree, come as prepackaged tools with Compaq's ATOM. Pixie counts basic block execution frequencies and can be useful for profile-based optimizations. Third Degree, like Purify, finds potential memory access errors in programs. The third tool we experimented with is a simple version of DIDUCE. As discussed in Section 2.1, DIDUCE monitors data accessed by instructions to look for anomalies in programs and help locate algorithmic errors. DIDUCE was originally implemented for Java, but we created a simple version with ATOM that tracks the values generated by load instructions. DIDUCE allows users to adjust the level of instrumentation according to their needs. To simulate different degrees of instrumentation, we tested two configurations: a heavy-weight version where every static load instruction in the program is instrumented, and a more lightweight version where only every 10th static load in the binary is instrumented. We will refer to the former experiment as DIDUCE and the latter as DIDUCE.1.

We applied these instrumentations to two programs: Vortex, an object-oriented database, and Perl, the PERL language interpreter, both from the SPECint benchmark suite. We used the training sets from CINT95 (specifically jumble.pl for Perl) for inputs.

Figure 3 shows an overview of how our monitoring examples performed. The first column shows the dynamic instruction counts before and after instrumentation, where the increases range from 2.6x when DIDUCE.1 is applied to Perl to 24x when DIDUCE is applied to Vortex. To see the effect on performance, we ran the programs on the SMT1/t1 machine where the slowdowns fall in a similar range, from 2.5x to 26.4x. When we increase the resources of the machine to the most generous SMT configuration but still execute with only one thread (SMT4/t1), the computations speed up by an average factor of 1.5x due to the additional ILP exploited. But by adding TLS features supporting eight thread contexts to the machine (SMT4/t8), we are able to obtain an *additional* 1.6x performance gain over the single-threaded SMT4 that has comparable resources. Overall, performance goes up by an average of 2.5x relative to SMT1/t1. Whereas DIDUCE.1 is originally slower by a factor of 2.5 and 3.5 for Perl and Vortex, the additional overhead is only 12% and 36% of the original computation on an SMT4/t8 machine, respectively. This kind of improvement makes using execution monitoring in production software a possibility. The heaviest instrumentation, DIDUCE on Vortex, takes only 11.2 times longer, down from more than 26 times. The tables also note the committed IPC of the individual runs, which on average have gone up from 2.2 to 5.4.

Also of interest is the performance of the programs with more limited SMT resources and varying numbers of thread contexts available. The overall IPCs achieved with the SMT2, SMT4* and SMT4 configurations with 1, 4 and 8 threads are shown in Figure 4. The base IPC with SMT1/t1 is also plotted as a single data point.

Program	Inst	Monitoring Slowdown			Speedups			IPC		
		SMT1/t1	SMT4/t1	SMT4/t8	4/1 vs. 1/1	4/8 vs. 1/1	4/8 vs.4/1	SMT1/t1	SMT4/t1	SMT4/t8
Vortex	3.1B	–	–	–	–	–	–	2.3	–	–
Pixie	19.5B	6.8	4.8	2.9	1.4	2.3	1.6	2.1	2.9	4.8
Third	55.0B	17.6	10.2	6.7	1.7	2.6	1.5	2.3	3.9	5.9
DIDUCE.1	10.8B	3.5	2.1	1.4	1.7	2.6	1.5	2.2	3.7	5.7
DIDUCE	75.7B	26.4	17.8	11.2	1.5	2.4	1.6	2.1	3.1	4.9
Perl	2.9B	–	–	–	–	–	–	2.0	–	–
Pixie	18.4B	5.5	3.8	1.9	1.5	2.9	2.0	2.3	3.4	6.7
Third	61.2B	19.4	13.7	9.7	1.4	2.0	1.4	2.2	3.1	4.4
DIDUCE.1	7.6B	2.5	1.6	1.1	1.6	2.3	1.4	2.1	3.3	4.8
DIDUCE	53.4B	16.2	10.0	5.5	1.6	3.0	1.8	2.3	3.7	6.8
Harmonic Mean					1.5	2.5	1.6	2.2	3.3	5.4

Figure 3: Summary of Vortex and Perl Monitoring

Most of the programs show limited performance improvement going from four to eight available threads, with the notable exception of the Pixie implementations of both Vortex and Perl where the threads are relatively short.

Some of the programs, like Third Degree applied to Vortex and the DIDUCE instrumentation of Perl have significant gains in performance when the four memory ports of SMT4* are increased to eight in SMT4. In our SMT architectures, however, there is a one-to-one correspondence between load-store queue ports and memory ports. Because additional memory ports are so costly, we experimented with an architecture that has eight load-store queue ports but only four memory ports, allowing load instructions that hit stores buffered in the load-store queues to issue more quickly than with SMT4*. We call this machine SMT4+ and tested it with eight threads. Figure 5 shows the resulting IPCs compared to the other SMT4 machines that have either four or eight memory and LSQ ports. The performance of SMT4+/t8 comes very close to that of SMT4/t8, suggesting that the SMT4+ architecture could be a very cost-effective design.

To further understand the performance of programs on our proposed architecture, we show in Figure 6 some internal statistics from the SMT4+/t8 runs. We show the average number of threads active each cycle, as well as what ultimately becomes of those threads: whether they are committed, evicted to make room for higher priority threads, or orphaned and deleted. Of the threads that do retire, we show what percentage had been restarted due to memory or register data violations before commit, as well as the number of instructions they executed and committed. For the cycle-based metrics, we distinguish between machine cycles and “thread cycles”, where *n* threads active during one machine cycle represents *n* thread cycles. Thus the LSQ full percentage expresses a likelihood that an active thread will find its own load-store queues full in any given cycle.

Programs that achieved the best speedups, such as Perl instrumented with either Pixie or DIDUCE, tend to have low violation rates and threads of consistent and moderate length. Third Degree applied to Perl, on the other hand, results in widely varying thread lengths and thus load imbalance, with only 42% of forked threads ultimately committing. In that benchmark there are also a significant number of register violations, the bulk of which are return values that were not correctly predicted. Both Pixie and DIDUCE applied to Vortex seem to suffer from rather poor front-end performance, fetching from the maximum two threads only 33% and 25% of the time and having the highest instruction cache miss rate among our benchmarks.

Application	SMT4*/t8	SMT4+/t8	SMT4/t8
Vortex			
Pixie	4.2	4.7	4.8
Third	4.8	5.5	5.9
DIDUCE.1	4.9	5.6	5.7
DIDUCE	3.9	4.9	4.9
Perl			
Pixie	5.9	6.7	6.7
Third	4.2	4.3	4.4
DIDUCE.1	4.5	4.7	4.8
DIDUCE	5.4	6.7	6.8

Figure 5: Effects of Memory vs. LSQ Ports on IPC

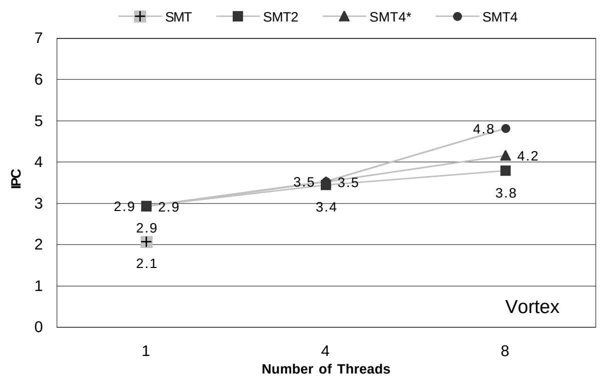
5.2 Speculative Transactions

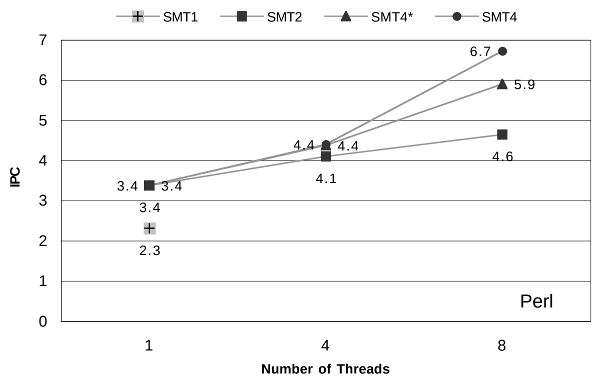
To demonstrate the ability of our proposed machine to monitor for buffer overruns and recover from them using transactions, we examined some sample vulnerabilities in common UNIX programs. We use detection methods based on two tools designed to catch these types of errors: Libsafe[31] and StackGuard[6]. Libsafe is a library of replacement versions for unsafe C string library functions, while StackGuard instruments the run-time stack to check if the return addresses have been tampered with.

Our first example is an ftp daemon implementation `bftpd`, version 1.0.22. The program has a vulnerability when processing an ftp `chown` command, due to a call to `sscanf()` from the `cmd_chown()` routine. We wrap the `cmd_chown()` routine into a transaction and return a failure code on if the transaction aborts. The monitoring code we use to detect this error is a modified version of `sscanf()` from Libsafe. Before executing the potentially dangerous code, it does a backwards traversal of the stack and copies frame pointer and return address values into its own private storage. It then executes the `sscanf()` and traverses the stack again to see if any frame pointers or return address were overwritten. If so, the transaction is aborted.

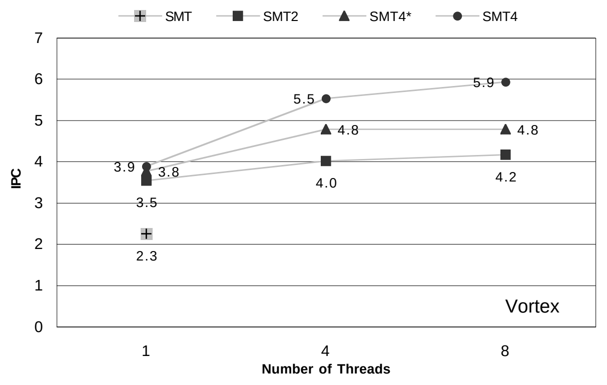
The second example is `imapd`, the IMAP daemon from the Pine 4.00 distribution. It contains a call to `strcpy()` from the `mail_auth()` function that can potentially cause an overflow. We wrap the `mail_auth()` function to create our transaction, and in case of an abort we return NULL. The detection code we use is again from Libsafe, but in this case the stack traversal only needs to be done once. Before executing the real `strcpy()`, the monitoring code locates which particular stack frame contains the des-

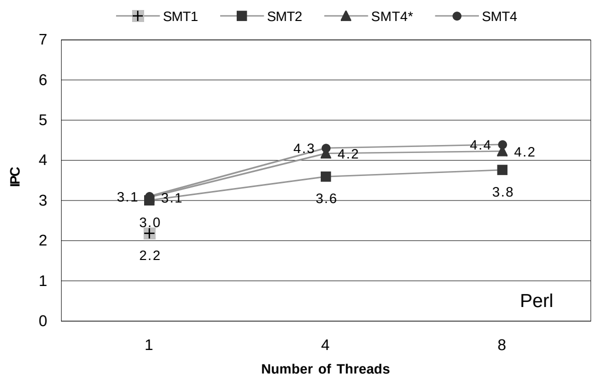
Pixie



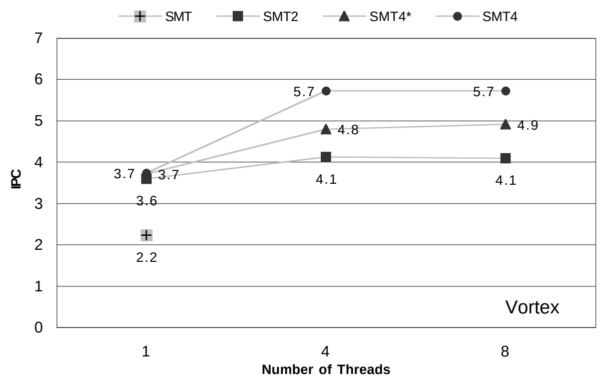


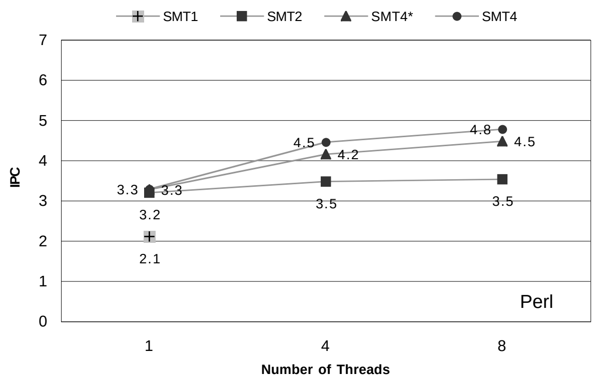
Third Degree



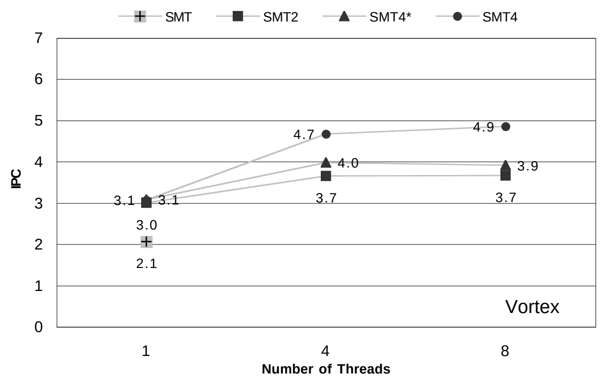


DIDUCE.1





DIDUCE



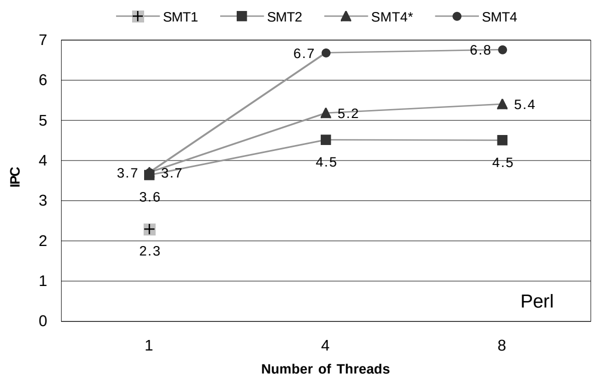


Figure 4: Performance of Execution Monitoring on the Proposed Architectures

Statistics	Pixie		Third		DIDUCE.1		DIDUCE	
	Vortex	Perl	Vortex	Perl	Vortex	Perl	Vortex	Perl
average # of active threads	4.9	6.5	4.5	6.2	3.2	3.7	3.6	5.6
% of threads								
committed	82	80	70	42	89	67	94	85
evicted	7	13	4	24	1	1	1	5
deleted	11	8	26	34	10	32	5	9
% of committed threads with								
memory violation	2.2	1.6	6.2	0.8	2.3	4.5	0.1	0.0
register violation	0.0	0.0	0.7	12.5	0.0	0.0	0.0	0.0
% of committed threads with								
0-10 instructions	0	0	0	10	0	0	0	0
11-25 instructions	10	7	1	12	0	0	0	0
26-50 instructions	86	91	10	19	0	0	0	0
51-100 instructions	4	2	35	15	40	21	92	98
101-200 instructions	1	0	44	13	59	70	6	2
201-500 instructions	0	0	10	29	1	9	1	0
% of thread cycles with								
LSQ full	1	1	7	3	12	17	4	4
% of machine cycles with								
0 fetches	29	12	12	6	22	12	42	17
1 fetch	38	31	26	15	42	33	33	35
2 fetches	33	56	56	80	36	55	25	48
% Icache miss rate	0.84	0.19	0.47	0.15	0.58	0.13	0.93	0.50

Figure 6: SMT4+/t8 Statistics

tination buffer. If the length of the source string is greater than the size of that frame, an error is signaled and we abort the transaction.

Our final example is `ntpd`, the network time protocol daemon, version 3.1.1. In the `ctl_getitem()` routine, which processes user input strings, there is a bug in the string manipulation code that can result in an overrun of a static buffer. We detect this error using a process much like StackGuard instrumentation. We manually add guard locations around the static variables of the procedure, and initialize them to known values at the beginning of the call. After the body of the procedure has executed, but before it returns, we check these guard locations to see if the values have been overwritten. If so, we abort and return NULL. If the values are correct, we commit the transaction before returning.

We have validated our implementation on our simulator, and measured the amount of state that was buffered in each transaction, as shown in Figure 7. First we show the natural amount of data generated by the original code that is wrapped by executing it with native C library functions and no instrumentation. The next two experiments show the amount of state generated in the monitor-and-recover version of the benchmarks for both valid inputs that allow the transaction to commit, as well as an exploit input that causes a buffer overrun and aborts the transaction.

All of the transactions fit into our default 4-way 32k L1 data cache. A 2-way 32k L1 unfortunately cannot buffer `bftpd-inst` with valid inputs, but that is due to our unoptimized Libsafe implementation. Libsafe normally dereferences the frame pointer chain to traverse the stack, but that did not work for the Alpha code we examined. The pointers were stored at varying offsets in the frame and did not form a linked list. Instead we used the `unwind()` and `exc_find_frame_ptr()` Alpha system functions to traverse the stack, but these routines are part of a multipurpose library and have expensive implementations. Sample calls to these functions alone generate up to 1,100 bytes of written data, occupying 42 cache lines. While these routines were sufficient for a demonstration of transaction functionality, real-life transaction support would clearly require a more efficient implementation.

Application	Input	Transactional State		
		bytes	cache lines	# of stores
ntpd	valid	92	5	15
ntpd-inst	valid	132	8	26
ntpd-inst	exploit	312	13	227
bftpd	valid	772	30	294
bftpd-inst	valid	3,248	132	4,010
bftpd-inst	exploit	2,968	113	12,489
imapd	valid	40	2	10
imapd-inst	valid	1,612	77	1,122
imapd-inst	exploit	1,796	82	1,156

Figure 7: Buffer Overrun Transactions

6. RELATED WORK

As we discussed in Section 4, speculative thread-level parallelism has been proposed by many different researchers, typically for the purposes of speeding up sequential integer programs[1, 5, 10, 19, 22, 25, 29] or floating point codes with dynamic dependences[3, 9, 28]. While many of the proposed machines could support the programming models we have described, our base architecture most closely resembles the DMT processor[1], which is an extension of the Simultaneous Multithreading machine proposal[33]. The DMT speculatively parallelizes ordinary binaries without recompilation or software support. The DMT does not have an explicit value predictor, so it is limited to spawn prediction for register. The DMT does trace buffering of the speculative thread execution which allows for selective recovery of only those instructions which must be re-executed due to data dependences. While this is obviously preferable, performance wise, to our scheme where all instructions in a thread must be re-executed, the implementation costs of this trace re-execution appear to be quite high. The DMT speculates on procedure continuations and loop continuations (the code after a loop), while we discuss only procedures here. Loop speculation is not relevant to the instrumentation code and transactions we have discussed here.

Transactional memory implementations have been proposed using a variety of implementations: hardware[13, 26], software[24], and combinations thereof[17]. Rollback and recovery in large-scale distributed systems have been studied extensively[8].

Our goal is to support fine-grained transactions with little or no overhead so they can become prevalent in everyday code. Speculative thread-level parallelism is being considered by many microprocessor designers, and chips utilizing it have already been released[30]. If chips implementing TLS become widely deployed, we feel it would be an excellent opportunity to leverage that functionality to deliver additional usability and reliability without much additional cost.

Executing "subordinate" code (such as prefetching code or exception handlers) in separate threads has been suggested by a number of researchers to improve performance, but those threads never create any committed side effects[36].

Patil and Fischer have proposed Shadow Processing to hide the overheads of run-time checking by executing the checks in a separate program on another processor[20]. This shadow program is generated by a source-to-source tool which slices the original program into a reduced form containing only the instructions needed for checking. Any non-reproducible computations, such as user inputs or system calls, are forwarded from the original to the shadow. They find that the resulting overheads in the original program are typically below 10%, but the shadow process can take up to ten times longer to complete. Of course, the continued execution of the main program is not recoverable. Thus, unlike our scheme, it is not generally possible for the checker to interact with or even stop the main process before serious errors or data corruption have occurred. For non-interactive "passive" instrumentation, however, there is a clear performance benefit from the user's perspective.

There have been proposals to use speculation to provide efficient support for locking. Martinez and Torellas propose buffering the states of critical sections in the processor cache to allow that code to execute while waiting in the background for the lock to be made available[18]. This can bring usability benefits to the program writer in that they can use coarser-granularity locks without suffering the usual performance degradation that results. Rajwar and Goodman suggest speculative lock elision, where lock operations are speculatively eliminated[21]. The on-chip write buffers are made part of the coherent state in order to allow for effectively atomic updates.

7. CONCLUSIONS

This paper advocates the use of a monitor-and-recover programming paradigm to create more reliable software, and proposes an architectural design that allows the software and hardware to cooperate in making this paradigm more efficient and easier to program.

Programmers simply write monitoring functions assuming the normal sequential execution semantics. For recovery, routines that may not complete properly are wrapped as "transactions" whose side effects can be discarded or committed as a whole. Our proposed TRY...ABORT...CATCH syntax for transactions is similar to that of an exception-handling construct, but it is much easier for programmers to deal with because all side effects, including register and memory updates, are discarded after an abort operation.

To support this monitor-and-recover paradigm efficiently, our proposed architecture gives software the control over the basic hardware mechanisms originally designed for thread-level speculation. We let the software specify that procedural speculation should be applied to specific monitoring functions. Because of the relative independence between the original code and the instrumentation, speculative parallelism is likely to be effective, thus hiding much of the costs of monitoring. The machine still guarantees sequential programming semantics, so exceptions or corrective actions can be handled accurately and safely should the monitoring function fail. We also show how our transaction construct can be translated directly into machine instructions that control when speculation begins and when to abort or commit the speculative state.

Our experiments with four different examples of execution monitoring suggest that TLS hardware can greatly reduce the overhead of monitoring–by a factor of 1.6 over a very wide superscalar with similar execution resources. Our average IPC of 5.4 is also much greater than typical results obtained when thread-level speculation is applied to ordinary sequential programs. We also show that the concept of fine-grain transactional programming is useful in catching buffer overrun problems through a number of real-life examples.

8. ACKNOWLEDGMENTS

The authors wish to thank Kostantinos Roussos for helpful discussions on programming practices. This material is based upon work supported in part by the National Science Foundation under Grant No. 0086160.

9. REFERENCES

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236, November 1998.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, Computer Sciences Department, University of Wisconsin-Madison, 1996.

[3] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 13–24, June 2000.

[4] L. Codrescu and D. S. Wills. On dynamic speculative thread partitioning and the MEM-slicing algorithm. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 40–46, October 1999.

[5] L. Codrescu, D. S. Wills, and J. D. Meindl. Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Transactions on Computers*, 50(1):67–82, 2001.

[6] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, Q. Zhang P. Wagle, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, January 1998.

[7] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, April 2001.

[8] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of

Computer Science, Carnegie Mellon University, October 1996.

[9] R. J. Figueiredo and J. Fortes. Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, September 2001.

[10] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[11] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.

[12] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, December 1992.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[14] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[16] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[17] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[18] J. F. Martínez and J. Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Proceedings of the Workshop on Memory Performance Issues at the 28th International Symposium on Computer Architecture*, June 2001.

[19] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, October 1999.

[20] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, 1995.

[21] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[22] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, November 1997.

[23] Y. Saito and B. Bershad. A transactional memory service in an extensible operating system. In *Proceedings of the 7th USENIX Security Conference*, pages 53–64, January 1998.

[24] N. Shavit and D. Touitou. Software transactional memory. In *Proceeedings of the Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[25] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 415–425, June 1995.

[26] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. Dept. of Computer Sciences Technical Report CS-TR-2000-1420, University of Wisconsin-Madison, October 2000.

[27] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[28] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–24, June 2000.

[29] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Conference on High-Performance Computer Architecture*, pages 2–13, January 1998.

[30] Sun. MAJC architecture tutorial. Technical report, Sun Microsystems Inc., 1999.

[31] T. Tsai and N. Singh. Libsafe 2.0: Detection of format string vulnerability exploits. White paper, Avaya Labs, February 2001.

[32] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Conference on High-Performance Computer Architecture*, pages 14–23, January 1998.

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

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

[35] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.

[36] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 219–229, November 1999.