Detailed Design and Evaluation of Redundant Multithreading Alternatives*

Shubhendu S. Mukherjee
VSSAD
Massachusetts Microprocessor Design Center
Intel Corporation
334 South Street, SHR1-T25
Shrewsbury, MA 01545
Shubu.Mukherjee@intel.com

Michael Kontz
Colorado VLSI Lab
Systems & VLSI Technology Operations
Hewlett-Packard Company
3404 East Harmony Road, ms 55
Fort Collins, CO 80525
michael_kontz@hp.com

Steven K. Reinhardt
EECS Department
University of Michigan, Ann Arbor
1301 Beal Avenue
Ann Arbor, MI 48109-2122
stever@eecs.umich.edu

ABSTRACT

Exponential growth in the number of on-chip transistors, coupled with reductions in voltage levels, makes each generation of microprocessors increasingly vulnerable to transient faults. In a multithreaded environment, we can detect these faults by running two copies of the same program as separate threads, feeding them identical inputs, and comparing their outputs, a technique we call Redundant Multithreading (RMT).

This paper studies RMT techniques in the context of both single- and dual-processor simultaneous multithreaded (SMT) single-chip devices. Using a detailed, commercial-grade, SMT processor design we uncover subtle RMT implementation complexities, and find that RMT can be a more significant burden for single-processor devices than prior studies indicate. However, a novel application of RMT techniques in a dual-processor device, which we term chip-level redundant threading *(CRT), shows higher performance than lockstepping the two cores, especially on multithreaded workloads.*

1. INTRODUCTION

Modern microprocessors are vulnerable to transient hardware faults caused by alpha particle and cosmic ray strikes. Strikes by cosmic ray particles, such as neutrons, are particularly critical because of the absence of any practical way to protect microprocessor chips from such strikes. As transistors shrink in size with succeeding technology generations, they become individually less vulnerable to cosmic ray strikes. However, decreasing voltage levels and exponentially increasing transistor counts cause overall chip susceptibility to increase rapidly. To compound the problem, achieving a particular failure rate for a large multiprocessor server requires an even lower failure rate for the individual microprocessors that comprise it. Due to these trends, we expect fault detection and recovery techniques, currently used only for mission-critical systems, to become common in all but the least expensive microprocessor devices.

One fault-detection approach for microprocessor cores, which we term *redundant multithreading* (RMT), runs two identical copies of the same program as independent threads and compares their outputs. On a mismatch, the checker flags an error and initiates a hardware or software recovery sequence. RMT has been proposed as a technique for implementing fault detection efficiently on top of a simultaneous multithreaded (SMT) processor (e.g., [18], [17], [15]). This paper makes contributions in two areas of RMT. First, we describe our application of RMT techniques to a processor that resembles a commercial-grade SMT processor design. The resulting design and its evaluation are significantly more detailed than previous RMT studies. Second, we examine the role of RMT techniques in forthcoming dual-processor single-chip devices.

Our implementation of the single-processor RMT device is based on the previously published simultaneous and redundantly threaded (SRT) processor design [15] (Figure 1a). However, unlike previous evaluations, we start with an extremely detailed performance model of an aggressive, commercial-grade SMT microprocessor resembling the Compaq Alpha Araña (a.k.a. 21464 or EV8) [12]. We call this our base processor. We found several subtle issues involved in adding SRT features to such a base SMT design. For example, adapting the SRT branch outcome queue, which uses branch outcomes from one thread (the "leading" thread) to eliminate branch mispredictions for its redundant copy (the "trailing" thread), to our base processor's line-prediction-driven fetch architecture proved to be a particularly difficult task. We also describe and analyze a simple extension to the proposed SRT design, called *preferential space redundancy,* which significantly improves coverage of permanent faults.

We then compare the performance of our SRT implementation with the baseline processor using the same detailed performance model. Our results indicate that the performance degradation of RMT (running redundant copies of a thread) compared to our baseline processor (running a single copy of the same thread) is 32% on average, greater than the 21% indicated by our previous work [15]. We also find that store queue size has a major impact on SRT performance. Our SRT implementation lengthens the average lifetime of a leading-thread store by roughly 39 cycles, requiring a significantly greater number of store queue entries to avoid stalls. We propose the use of per-thread store queues to increase the number of store queue entries without severely impacting cycle time. This optimization reduces average performance degradation from 32% to 30%, with significant benefits on several individual benchmarks.

We also expand our performance study beyond that of previous work by examining the impact of SRT on multithreaded workloads. We run two logical application threads as two redundant thread pairs, consuming four hardware thread contexts on a single processor. We find our SRT processor's performance degradation for such a configuration is about 40%. However, the use of per-thread store queues can reduce the degradation to about 32%.

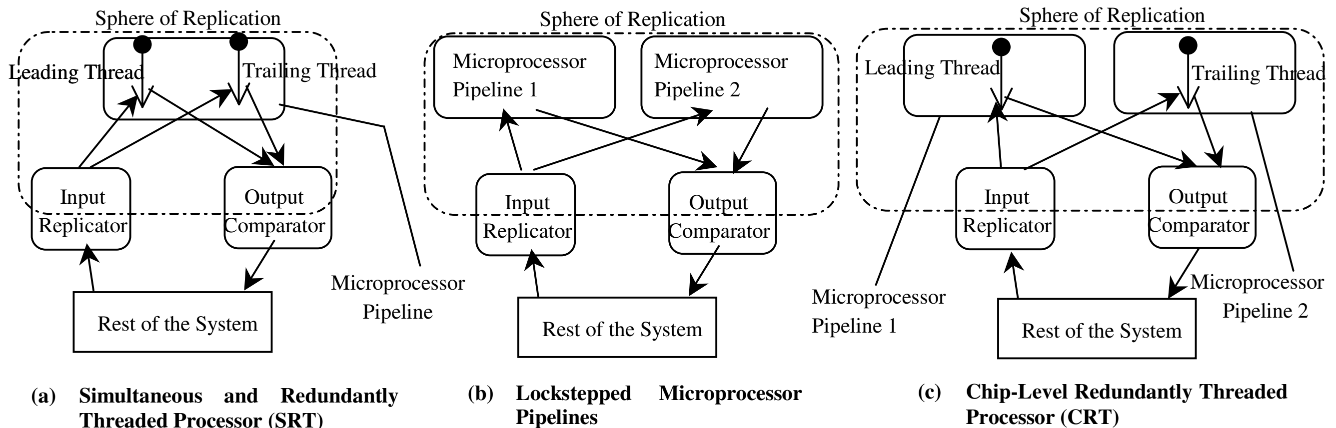
Our second area of contribution involves the role of RMT techniques in dual-processor single-chip devices. Initial examples of these two-way chip multiprocessors (CMPs) are shipping (e.g., the IBM Power4 [7] and the HP Mako [8]). We expect this configuration to proliferate as transistor counts continue to grow exponentially and wire delays, rather than die area, constrain the size of a single processor core.

A two-way CMP enables on-chip fault detection using *lockstepping*, where the same computation is performed on both processors on a cycle-by-cycle basis, that is, in "lockstep" (Figure 1b). Lockstepping has several advantages over SRT-style redundancy. Lockstepping is a well-understood technique, as it has long been

* This work was performed at Compaq Computer Corporation, where Shubhendu S. Mukherjee was a full-time employee, Michael Kontz was an intern, and Steven K. Reinhardt was a contractor.

1063-6897/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY



(a) Simultaneous and Redundantly Threaded Processor (SRT) **(b) Lockstepped Microprocessor Pipelines** **(c) Chip-Level Redundantly Threaded Processor (CRT)**

Figure 1. Fault Detection Using SRT, Lockstepped Microprocessors, and CRT. Specifically, in our implementations, the microprocessor pipelines, input replicators, and output comparators are on the same chip. The "rest of the system" is split into on-chip components (L2 cache, memory controllers, on-chip router) and off-chip components (memory, disks, other I/O devices).

used across separate chips on commercial fault-tolerant systems (e.g., Compaq Himalaya systems [30]), and is used on-chip in some fault-tolerant processors (e.g., the IBM G5 [21]). Lockstepping also provides more complete fault coverage than SRT, particularly for permanent faults, as redundant computations are executed on physically separate hardware. However, lockstepping uses hardware resources less efficiently than SRT, as both copies of a computation are forced to waste resources—in lockstep—on misspeculation and cache misses. Lockstepping also requires all signals from both processors to be routed to a central checker module before being forwarded to the rest of the system, increasing cache-miss latencies.

To combine the fault coverage of lockstepping with the efficiency of SRT, we propose a new technique—*chip-level redundant threading (CRT)*—that extends SRT techniques to a CMP environment (Figure 1c). As in SRT, CRT uses loosely synchronized redundant threads, enabling lower checker overhead and eliminating cache miss and misspeculation penalties on the trailing thread copy. As in lockstepping, the two redundant thread copies execute on separate processor cores; they are not multiplexed as different thread contexts on a single core as in SRT.

On single-thread workloads, CRT performs similarly to lockstepping, because the behavior of CRT's leading thread is similar to that of the individual threads in the lockstepped processors. However, with multithreaded workloads, CRT "cross-couples" the cores for greater efficiency. For example, with two application threads, each core runs the leading thread for one application and the trailing thread for the other. The resources freed up by CRT on each core from optimizing one application's trailing thread are then applied to the more resource-intensive leading thread of a different application. On these multithreaded workloads, CRT outperforms lockstepping by 13% on average, with a maximum improvement of 22%.

The rest of the paper is organized as follows. The next two sections discuss background material: Section 3 describes the specifics of the previously proposed SRT scheme and Section 4 briefly describes our baseline processor architecture. Section 5 begins our contributions, describing our adaptation of SRT concepts to our baseline processor model. Section 6 covers our new chip-level redundant threading (CRT) technique for two-way CMP devices. We describe our evaluation methodology in Section 7 and present results in Section 8. We conclude in Section 9.

2. BASIC SRT CONCEPTS

An SRT processor detects faults by running two identical copies of the same program as independent threads on an SMT processor [15]. For several reasons, it is useful to maintain one thread slightly further along in its execution than the other, creating a distinct *leading* thread and *trailing* thread within the pair. Adding SRT support to an SMT processor involves two key mechanisms: *input replication* and *output comparison*. Input replication guarantees that both threads see identical input values; otherwise, they may follow different execution paths even in the absence of faults. Output comparison verifies that the results produced by the two threads are identical before they are forwarded to the rest of the system, signaling a fault and possibly initiating a system-dependent recovery process if they are not identical.

A key concept introduced in the SRT work is the *sphere of replication*, the logical boundary of redundant execution within a system. Components within the sphere enjoy fault coverage due to the redundant execution; components outside the sphere do not, and therefore, must be protected via other means, such as information redundancy. Values entering the sphere of replication are inputs that must be replicated; values leaving the sphere of replication are outputs that must be compared.

For this paper, we chose the larger of the spheres of replication described in the SRT paper, including the processor pipeline and register files, but excluding the L1 data and instruction caches. Sections 2.1 and 2.2, respectively, review the required input replication and output comparison mechanisms for this sphere. Section 2.3 reviews two SRT performance optimization techniques: *slack fetch* and the *branch outcome queue*.

2.1 Input Replication

Because our sphere of replication excludes the L1 data and instruction caches, we must perform input replication on values coming from these structures, i.e., the results of cacheable loads and instruction fetches. For cached load value replication, SRT proposes a first-in first-out *load value queue*. As each leading-thread load retires, it writes its address and load value to the load value queue. The trailing thread's loads read the load value queue in program order, verifying the load address and retrieving the data. Because the data is not read redundantly out of the cache, the load value queue contents must be protected by some other means, e.g., ECC. The load value queue prevents external updates to shared

IEEE COMPUTER SOCIETY

Table 1. Base Processor Parameters

IBOX	Fetch Width	2 8-instruction chunks (from same thread) per cycle
	Line Predictor	Predicts two chunks per cycle. The two chunks can be non-sequential Total number of entries = 28K
	L1 Instruction Cache	64 Kbytes, 2-way set associative with way prediction, 64 byte blocks
	Branch Predictor	208 Kbits
	Memory Dependence Predictor	Store Sets, 4K entries [2]
	Rate Matching Buffer	Collapses 2 8-instruction chunks (from same thread) to create one map chunk with up to 8 instructions
PBOX	Map Width	One 8-instruction chunk (from same thread) per cycle
QBOX	Instruction Queue	128 entries window
	Issue Width	8 instructions per cycle
RBOX	Register File	512 physical registers, 256 architectural registers (64 registers per thread)
EBOX & FBOX	Functional Units (includes a register cache)	8 operations per cycle 8 integer units, 8 logic units, 4 memory units, 4 floating point units
MBOX	L1 Data Cache	64 Kbytes, 2-way set associative, 64 byte blocks, 3 load ports, one write port
	Load Queue	64 entries
	Store Queue	64 entries
	Coalescing Merge Buffer	16 64-byte blocks
System Interface	L2 Cache	3 Megabytes, 8-way set-associative, 64 byte cache blocks
	Memory Interface	2 Rambus controllers, 10 Rambus channels
	Network Router & Interface	On-chip two-dimensional mesh router, similar to the Alpha 21364 [10]

memory locations—e.g., from other processors or I/O devices—from causing the threads to diverge.

As in the original SRT work, we assume that the instruction space is read-only. Thus, as long as both threads generate the same sequence of program counter values (which they will in the absence of faults), they will receive the same instruction values from the memory system. Most modern systems already require special synchronization operations to modify the instruction space safely; these can be extended if necessary to provide proper replication of instruction values.

Techniques for replicating less frequent inputs, such as uncached load values and interrupt inputs, are also described in the original SRT paper. We leave detailed implementation of these replication mechanisms for future work.

2.2 Output Comparison

Cacheable stores are the primary form of output from our sphere of replication. SRT proposes an enhanced store queue for output comparison of cacheable stores. Leading-thread store addresses and data are held in the store queue until they are verified by comparing the address and data values generated by the corresponding store in the trailing thread. Once a store has been matched and verified, a single store operation is forwarded outside the sphere. Again, we defer implementation of other required, but less frequent, output comparisons identified in the SRT paper, such as uncached stores and uncached load addresses, to future work.

Figure 2. Base Processor's Integer Pipeline. I = IBOX consisting of the thread chooser, line prediction, instruction cache access, and the rate matching buffer. P = PBOX consisting of wire delays and register rename, Q = QBOX consisting of instruction queue operations, R = RBOX consisting of register read stages, E = EBOX consisting of functional units, and M = MBOX consisting of data caches, load queue, and store queue. In our evaluation, we assumed the following latencies: I = 4, P = 2, Q = 4, R = 4, E =1, and M = 2 cycles.

2.3 Performance Optimizations

An SRT processor lends itself to a few significant performance optimizations. The key insight is that the trailing thread can use information from the leading thread's execution to make its own execution more efficient.

One such situation occurs when the leading thread encounters an instruction cache miss. If the trailing thread is sufficiently delayed, then its corresponding fetch may not occur until the block has been loaded into the instruction cache, avoiding a stall. A similar benefit can be obtained on data cache misses; even though the trailing thread reads load values out of the load value queue, a sufficient lag between the threads will guarantee that the load value queue data will be present when the trailing thread's load executes, even if the corresponding leading-thread access was a cache miss. The original SRT work proposed a *slack fetch* mechanism to achieve this benefit. In slack fetch, instruction fetch of the trailing thread is delayed until some number of instructions from the leading thread has retired, forcing a "slack" between the threads to absorb these cache delays. Our earlier work [15] showed that slack fetch could achieve about a 10% boost in performance on average.

A second optimization uses the result of leading-thread branches to eliminate control-flow mispredictions in the trailing thread. To achieve this effect, SRT proposes a *branch outcome queue*, a simple FIFO that forwards branch and other control flow targets from the leading thread's commit stage to the trailing thread's fetch stage.

3. BASE PIPELINE

Our base processor is an eight-way superscalar SMT machine with four hardware thread contexts. Table 1 lists some of the architectural parameters of our base processor used in this paper.

Figure 2 shows the base processor's pipeline. The pipeline is divided into the following segments: IBOX (instruction fetch), PBOX (instruction rename), QBOX (instruction queue), RBOX (register read), EBOX & FBOX (integer and floating point functional units), and MBOX (memory system). There are additional cycles incurred to retire instructions beyond the MBOX. Below we describe our base processor architecture's specific portions and boxes, which we modified to create an SRT architecture.

3.1 IBOX

The IBOX fetches instructions in 8-instruction "chunks" and forwards them to the instruction rename unit or PBOX. In each cycle, the IBOX fetches up to 16 instructions (two chunks) from a single thread.

In our evaluation, we assume a four-stage IBOX. The first stage chooses the thread for which instructions will be fetched in each cycle. We pick the thread with the minimum number of instructions in its rate-matching buffer, giving an approximation of the ICOUNT fetch policy described by Tullsen, et al [28]. The second stage uses the line predictor to predict two chunk addresses per

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

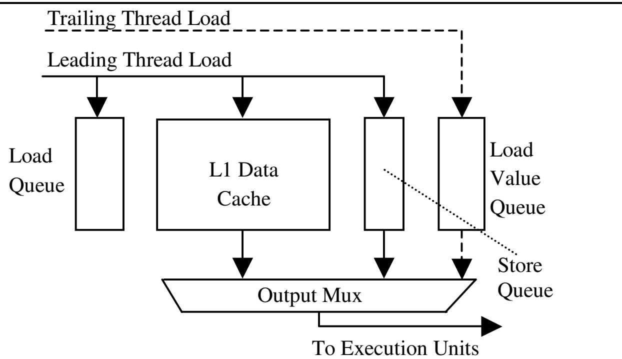


Figure 3. The load value queue integrated in the MBOX.

cycle. As in the Alpha 21264, our base processor's line predictor generates a sequence of predicted instruction-cache line (set and way) indices to drive instruction fetch. The third stage uses these addresses to fetch two potentially non-contiguous chunks of eight instructions each from the instruction cache. In the fourth stage, the processor writes these instructions to the per-thread rate matching buffers. Control-flow predictions from the branch predictor, jump target predictor, and return address stack become available in this stage. If these predictions disagree with the line prediction, the line predictor is retrained and the fetch is re-initiated.

In our base processor, the line predictor is indexed by the predicted line index from the previous cycle. Thus the line predictor must be "primed" with a line prediction from an external source to begin predicting down a different instruction stream, e.g., when the thread chooser decides to switch to a different thread. The thread chooser stores the line predictor's last predictions for each thread and reissues those predictions whenever fetching for a thread is suspended and restarted. The task of selecting either the line predictor's output or the thread chooser's stored prediction to use as the instruction cache index falls on the instruction cache address driver, which is located in the second IBOX stage after the line predictor. There exists a complex interaction between the thread chooser, line predictor, and address driver to facilitate instruction fetch. Section 4.4 discusses how we overcame this complexity to mimic the functionality of SRT's branch outcome queue in this line-prediction-oriented fetch scheme.

3.2 PBOX

The PBOX performs initial processing on instructions fetched by the IBOX, including register renaming and partial decoding. The PBOX also maintains checkpoints of various mapping tables to facilitate fast recovery from branch mispredictions and other exceptional conditions. In our evaluation, we assume that the PBOX occupies two pipeline stages.

3.3 QBOX

The QBOX receives instructions in program order from the PBOX and issues them out of order to the EBOX, FBOX or MBOX when their operands become ready. The QBOX also retires instructions, committing their results to architectural state in program order. The QBOX contains three main structures: the instruction queue, which schedules and issues instructions from the PBOX; the in-flight table, which tracks instructions from issue until they complete execution; and the completion unit, which tracks them up to retirement.

The instruction queue is the most complex part of the QBOX. It holds 128 instructions, and accepts and issues up to eight instructions per cycle. The queue is divided into upper and lower halves, each with 64 entries. Both halves have the same capabilities (in terms of functional units) and each can issue up to four instructions per cycle. An instruction is assigned to a particular queue half based on its position in the chunk created by the rate-matching buffer.

3.4 MBOX

The MBOX processes memory instructions, such as loads, stores, and memory barriers. The QBOX can issue a maximum of four memory operations per cycle, with a maximum of two stores and three loads.

Loads entering the MBOX record themselves in the load queue, probe the data cache and store queue simultaneously, and return their result to the load queue in a single cycle. On a data cache or store queue hit, the data is forwarded in the next cycle to the execution units for bypassing to dependent instructions.

Similarly, stores entering the MBOX record themselves in the store queue and probe the load queue to check for store-load order violation. Store data arrives at the store queue two cycles after the store address. When a store retires, its address and data are forwarded to a coalescing merge buffer, which eventually updates the data cache.

For multithreaded runs, we divide up the 64-entry load and store queues statically among the different threads. Thus, when this base processor runs two threads, we allocate 32 entries to each thread. For multithreaded runs with four threads, we allocate 16 entries to each thread.

4. SRT ON OUR BASE PROCESSOR

One of the major goals of this work is to understand the feasibility and performance impact of implementing SRT extensions on a realistic, commercial-grade SMT processor. As discussed in Section 2, the key SRT mechanisms are input replication and output comparison; we discuss our implementation of these mechanisms on our base processor in Sections 4.1 and 4.2. Section 4.3 describes additional changes required to avoid deadlock situations. One of the greatest challenges in our effort was to map SRT's branch outcome queue to our base processor's line-predictor driven fetch architecture; the specific problems and our solutions are detailed in Section 4.4. Finally, Section 4.5 describes our implementation of *preferential space redundancy*, an enhancement to the original SRT proposal for improved fault coverage.

4.1 Input Replication

We use a variant of SRT's load value queue (LVQ), described in Section 2.1, to forward cached load values from the leading to the trailing thread. Figure 3 shows the relationship of the LVQ to the rest of the MBOX.

Leading-thread loads entering the MBOX probe the load queue, data cache, and store queue, as in the base design. The QBOX completion unit writes the address and data values generated by these loads to the LVQ as the loads retire. Trailing-thread loads bypass all three structures and directly access the LVQ. The LVQ forwards load values to the MBOX output mux, which now has one additional input. Because the QBOX can issue up to three loads per cycle, the LVQ must support three concurrent accesses.

The original SRT design assumes that the trailing thread issues its loads in program order, preserving the FIFO structure of the LVQ. Although we could easily add this constraint to the QBOX scheduler by creating dependences between loads, this approach would limit the trailing thread to issuing at most one load per cycle. Rather than modifying the scheduler to allow three "dependent"

IEEE COMPUTER SOCIETY

loads to issue to the LVQ in a single cycle, we modified the LVQ to allow out-of-order load issue from the trailing thread. Each leading-thread load is assigned a small load correlation tag value by the PBOX, which is written to the LVQ entry along with the load's address and data. Using the line prediction queue, described in Section 4.4, we can easily associate the same tag with the corresponding load from the trailing thread. When the trailing thread issues, it uses this tag to perform an associative lookup in the LVQ to read out the appropriate address and data values. Then, the LVQ entry is deallocated.

This LVQ probe is similar to the lookup done on each load address in the store queue; in fact, it is slightly simpler, since the LVQ need not worry about multiple address matches, partial forwarding cases, or relative ages of entries. Thus, an associative LVQ equal in size to the store queue should be able to support three simultaneous accesses per cycle without adding to the critical path.

The LVQ provides a small additional performance benefit because it keeps the trailing thread's loads out of the load queue, thereby freeing up entries for the leading thread's loads. Thus, when one application program runs in SRT mode with two redundant threads, the leading thread gets all 64 load-queue entries. Similarly, when two application programs run in SRT mode, each with two redundant threads, each leading thread gets 32 load queue entries.

4.2 Output Comparison

As described in Section 2.2, the original SRT proposal included an enhanced store queue to perform output comparison on cacheable stores. We implemented a separate structure, the *store comparator*, which sits next to the store queue and monitors its inputs. When a trailing-thread store instruction and its data enter the store queue, the store comparator searches itself for a matching entry from the leading thread. If a matching entry exists, the store comparator performs the store comparison and signals the store queue that the specific store is ready to be retired to the data cache.

Forcing every leading-thread store to wait in the store queue until the corresponding trailing-thread store is executed increases store-queue utilization significantly, causing additional pipeline stalls when the store queue becomes full. Section 7 shows that these stalls degrade SRT performance noticeably relative to the base architecture. However, because the data cache is outside our sphere of replication, the cache cannot be updated until this output comparison is performed. Meanwhile, subsequent leading-thread loads must compare against these waiting stores to obtain the latest memory values, just as they compare against speculative stores in the base architecture. Thus we see no practical alternative to leaving these stores in the store queue until they are checked.

Unfortunately, the store queue CAM is a critical path in our base architecture; increasing its size beyond 64 entries would force the CAM to be pipelined across multiple cycles, increasing the load-to-use delay and adversely affecting performance. Instead, we propose addressing this problem by creating separate per-thread store queues of 64 entries each. Although this proposal does not increase the size of any individual store queue CAM, the physical implementation may still be challenging because of the additional wires and multiplexors introduced in the path of the load probe. Nevertheless, our results in Section 7 do show that per-thread store queues can provide a significant boost in performance.

4.3 Avoiding Deadlocks

An SMT machine is vulnerable to deadlock if one thread is allowed to consume all of a particular resource (e.g., instruction queue slots) while stalled waiting for another thread to free a different resource or post a synchronization event. Our base design assumes that such inter-thread dependencies do not exist. While this assumption is valid for conventional multithreaded workloads, dependencies between leading and trailing threads in an SRT machine significantly increase the possibility of deadlock. In particular, the leading thread's dependence on seeing a corresponding store from the trailing thread before freeing a store-queue entry can lead to frequent deadlocks.

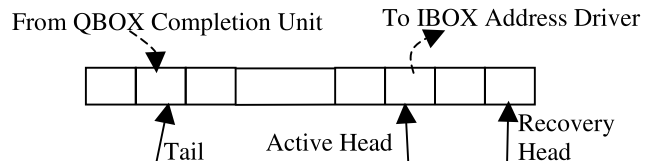


Figure 4. The Line Prediction Queue in the IBOX Line Prediction Unit.

For example, the leading thread can easily fill the QBOX instruction queue, backing up into the PBOX pipeline itself. In this case, the trailing thread is unable to move instructions from its IBOX rate-matching buffer into the PBOX. If the leading thread is stalled waiting for a matching trailing-thread store, deadlock results. To avoid this situation, we made all PBOX storage structures per thread, and reserved space for one chunk per thread in the QBOX queue. Because the IBOX already contained per-thread rate matching buffers, no changes were needed there.

A second class of deadlocks induced by our line prediction queue is covered in Section 4.4.2.

4.4 Implementing the Branch Outcome Queue

The original SRT design proposed a simple FIFO called the branch outcome queue, which forwards leading-thread branch targets to the trailing thread's fetch stage. Ideally, since this queue represents a perfect stream of target predictions (in the absence of faults), the trailing thread should never fetch a misspeculated instruction. Unfortunately, our base processor uses a line predictor to access the instruction cache; the branch and other control-flow predictors serve only to verify these line predictions. Because the line predictor's misprediction rate is significant (between 14% and 28% for our benchmarks), using the branch outcome queue in place of the branch target prediction structures, as originally proposed, would still allow a noticeable amount of misfetching on the trailing thread.

A simple alternative is to share the line predictor between the two redundant threads, in the hope that the leading thread would train the line predictor, improving prediction accuracy for the trailing thread. Unfortunately, this scheme does not work well due to excessive aliasing in the line prediction table.

Instead, we adapt the concept of the branch outcome queue to fit the base design, and use a *line prediction queue* to forward correct line predictions from the leading to the trailing thread. The line prediction queue provides perfect line predictions to the trailing thread in the absence of faults, thereby completely eliminating misfetches.

The implementation of the line prediction queue was quite challenging, and required careful consideration to maintain perfect line prediction accuracy and avoid deadlocks. We organize the design issues into two parts: those involved with reading predictions from the line prediction queue on the IBOX end and those regarding writing predictions to the line prediction queue on the QBOX end. Although the details of our design are specific to our base architecture, we believe many of the challenges we encoun-

IEEE COMPUTER SOCIETY

tered are inherent to most modern microarchitectures, which are not designed to generate or operate on a precise sequence of fetch addresses. We were certainly surprised that a seemingly straightforward feature, added easily to our earlier SimpleScalar-based simulator, induced such complexities in this more realistic design.

After incorporating the line prediction queue into our design, we found that the "slack fetch" mechanism of the original SRT design, described in Section 2.3, was not necessary. The inherent delay introduced by waiting for leading-thread retirement before initiating the corresponding trailing-thread fetch was more than adequate to provide the benefits of slack fetch. In fact, given the store-queue pressure discussed in Section 7.1, we found that the best performance was achieved by giving the trailing thread priority, and fetching based on the line prediction queue whenever a prediction was available.

4.4.1 Reading from the Line Prediction Queue

One set of line prediction queue challenges came about because the IBOX was not designed to expect a precise stream of line predictions. Specifically, as described in Section 3.1, the line predictor sends predictions to the address driver mux, but does not know whether the prediction is selected by the address driver or not. Even if the address driver selects the line predictor's output, the access may be a cache miss, requiring the prediction to be resent after the needed block is filled from the L2 cache.

A conventional line predictor is happy to repeatedly predict the same next line, given the same input (e.g. from the thread chooser's stored prediction latch). However, the line prediction queue stores a precise sequence of chunk addresses; if a prediction is read from the line prediction queue but not used, a gap will appear in the trailing thread's instruction stream.

We addressed this problem in two steps. First, we enhanced the protocol between the address driver and line predictor to include an acknowledgment signal. On a successful acceptance of a line prediction, the address driver acks the line predictor, which advances the head of the line prediction queue. Otherwise, the line prediction queue does not adjust the head, and resends the same line prediction on the subsequent cycle.

Instruction cache misses require additional sophistication; in these cases, the address driver does accept a line prediction from the line prediction queue, but must reissue the same fetch after the miss is handled. To deal with this and similar situations, we provided the line prediction queue with two head pointers (Figure 4). The *active head* is advanced by acks from the address driver, and indicates the next prediction to send. A second *recovery head* is advanced only when the corresponding instructions have been successfully fetched from the cache. Under certain circumstances, including cache misses, the IBOX control logic can request the line prediction queue to roll the active head back to the recovery head, and reissue a sequence of predictions.

4.4.2 Writing to the Line Prediction Queue

The tail end of the line prediction queue, where leading-thread instructions retiring from the QBOX generate predictions for the trailing thread, also presented some challenges. Each line prediction queue entry corresponds to a single fetch chunk, i.e., a contiguous group of up to eight instructions. To achieve reasonable fetch efficiency, the logic at the QBOX end of the line prediction queue must aggregate multiple retired instructions into a single chunk prediction. The key decision made by this logic is when to terminate a trailing-thread fetch chunk and actually record a prediction in the line prediction queue.

Some chunk-termination situations are clear, such as when two retiring instructions have non-contiguous addresses, or when the eight-instruction chunk limit has been reached. However, other necessary criteria were more subtle, and resulted in deadlocks when they were ignored. For example, a memory barrier instruction is not eligible to retire until all preceding stores have flushed from the store queue. If a store precedes a memory barrier in the same leading-thread fetch chunk, the store will not flush until the corresponding trailing-thread store executes. However, this trailing-thread store will not be fetched until the chunk is terminated and its address is forwarded via the line prediction queue. Under normal circumstances (in non-SRT mode), we will not terminate the chunk until the memory barrier retires, as it is a contiguous instruction and can be added to the current chunk. To avoid this deadlock situation, however, we must force termination of the trailing-thread fetch chunk whenever the oldest leading-thread instruction is a memory barrier.

A similar situation arises due to base processor's handling of partial data forwarding. For example, if a word load is preceded by a byte store to the same location, our base processor flushes the store from the store queue so that the load can pick up the full word from the data cache. If the store and load are in the same leading-thread fetch chunk, we must terminate the chunk at the store to allow the trailing-thread's store to be fetched. Then the store will be verified and exit the store queue, enabling the load to execute.

Interestingly, the line prediction queue logic can occasionally create trailing-thread fetch chunks that are larger than those of the leading thread. For example, a predicted taken branch in the middle of a leading-thread fetch chunk will cause the chunk to be terminated. If the branch was mispredicted, and actually fell through, we can add the fall-through instructions to the trailing thread's fetch chunk.

4.5 Improving Fault Coverage

The original SRT proposal focuses on detection of *transient* faults, which only temporarily disrupt processor operation. For example, a cosmic ray may strike a latch and change its stored value; this fault will last only until the latch is next written, at which time it will again have a correct state. However, microprocessors are also vulnerable to *permanent* faults, which can arise due to manufacturing defects or electromigration. A transient fault in a rarely written latch value (e.g., a mode bit written only at boot time) may also behave like a permanent fault. This section describes how we extended our SRT design to provide improved coverage of permanent faults with negligible performance impact.

The original SRT design's vulnerability to permanent faults is due to the combination of space redundancy (where redundant instructions use physically distinct hardware resources) and time redundancy (where redundant instructions use the same hardware resource at different times). Note that the same pair of instructions can be covered by space redundancy in some portions of the pipeline and time redundancy in others. Time redundancy provides effective coverage for transient and some timing-dependent faults, but is not effective for detecting permanent hardware faults.

We introduce a new technique, called *preferential space redundancy*, which simply biases an SRT processor to provide space redundancy rather than only time redundancy whenever the option exists. As a concrete example, we implemented preferential space redundancy for the QBOX instruction queue, providing extremely high coverage of permanent faults in this critical structure. We leverage our base processor's partitioning of the queue into upper and lower halves. When a leading-thread instruction executes, it records which half of the queue it traversed. We add the up-

IEEE COMPUTER SOCIETY

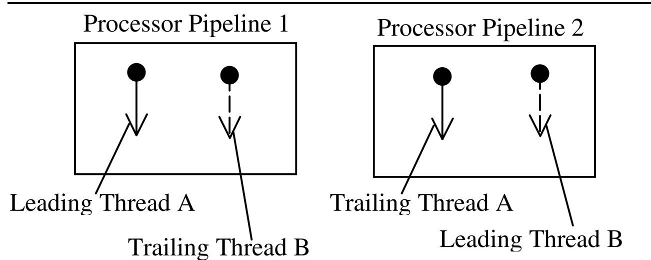


Figure 5. CRT processor configuration. The arrows represent threads in the processors. A and B are logically distinct programs. Each program runs a redundant copy: a leading copy and a trailing copy in the physically distinct processor pipelines. Note that the leading thread of A is coupled with the trailing thread of B and vice versa in the two processors.

per/lower selection bits for all instructions in a fetch chunk to each line prediction queue entry as it is forwarded to the trailing thread's fetch stage. These bits are carried along to the QBOX, which then assigns the corresponding trailing-thread instructions to the *opposite* half of the queue relative to their leading-thread counterparts. We thus guarantee that corresponding instructions are handled by distinct IQ entries. Section 7.1.1 will show that this technique virtually eliminates all time redundancy in the instruction queue and function units without any loss in performance.

Preferential space redundancy applies only to situations in which there are multiple identical structures to provide space redundancy, such as queue entries or function units. Fortunately, the remaining structures that provide only time redundancy are primarily transmission lines, which can be protected from single permanent faults by the addition of parity bits. We therefore believe that an SRT processor can be designed to detect most, if not all, single permanent faults, in addition to the transient faults for which it was originally designed.

5. CHIP-LEVEL REDUNDANT THREADING

In this section, we extend SRT techniques to the emerging class of chip multiprocessors (CMPs) to create a *chip-level redundantly threaded (CRT)* processor, achieving lockstepping's permanent fault coverage while maintaining SRT's low-overhead output comparison and efficiency optimizations. The basic idea of CRT is to generate logically redundant threads, as in SRT, but to run the leading and trailing threads on separate processor cores, as shown in Figure 5.

The trailing threads' load value queues and line prediction queues now receive inputs from leading threads on the other processor. Similarly, the store comparator, which compares store instructions from redundant threads, receives retired stores from the leading thread on one processor and trailing thread on another processor. Clearly, to forward inputs to the load value queue, line prediction queue, and the store comparator, we need moderately wide datapaths between the processors. We believe that the processor cores can be laid out on the die such that such datapaths do not traverse long distances. These datapaths will be outside the sphere of replication and must be protected with some form of information redundancy, such as parity.

CRT processors provide two advantages over lockstepped microprocessors. First, in lockstepped processors, all processor output signals must be compared for mismatch, including miss requests from the data and instruction caches. This comparison is in the critical path of the cache miss, and often adversely affects performance. More generally, the checker must interpose on every logical signal from the two processors, check for mismatch, and then forward the signal outside the sphere of replication. Of course, a CRT processor incurs latency to forward data to the line prediction queue, load value queue, or store comparator, but these queues serve to decouple the execution of the redundant threads and are not generally in the critical path of data accesses.

Second, CRT processors can run multiple independent threads more efficiently than lockstepped processors. By pairing leading and trailing threads of different programs on the same processor, we maximize overall throughput. A trailing thread never misspeculates, freeing resources for the other application's leading thread. Additionally, in our implementation, trailing threads do not use the data cache or the load queue, freeing up additional resources for leading threads.

Our evaluation, detailed in Section 7.2, shows that our CRT processor performs similarly to lockstepping for single-program runs, but outperforms lockstepping by 13% on average (with a maximum improvement of 22%) for multithreaded program runs.

6. METHODOLOGY

This section describes the performance model, benchmarks, target architecture parameters, and evaluation metric that we used for our evaluation.

6.1 Asim: The Performance Model Framework

Asim [4] is a performance model framework, which was used to build an extremely detailed performance model of a modern, aggressive, dynamically scheduled, eight-wide SMT processor resembling the Alpha Araña processor. This model provides cycle-by-cycle simulation of most of the components in this base processor. We modified this detailed base processor model written in Asim to create the SRT and the CRT processor models.

6.2 Benchmarks

For our evaluation with single programs, we used all the 18 SPEC CPU95 benchmarks (http://www.spec.org). To get to the interesting portions of the benchmarks, we skipped between 250 million and 2 billion instructions. Then, we warmed up the processor structures for one million instructions and executed 15 million committed instructions for each program.

For our evaluation with multiple programs, we combined a subset of the SPEC CPU95 programs. For runs with two programs, we combined two of *gcc*, *go*, *fpppp*, and *swim*. The four benchmarks generate a total of six pairs. Similarly, for our four-program runs, we combined four of *gcc*, *go*, *ijpeg*, *fpppp*, and *swim* to generate a total of 15 combinations. We ran 15 million committed instructions for each program. Thus, our CRT processor running four logical threads executed a total of 120 million instructions (15 million instructions per program × 4 programs × 2 redundant threads per program).

We did not use the SPEC CPU2000 suite partly due to limitations in our simulation environment, and partly to allow us to compare our single-program SRT results with our earlier evaluation [15], which also used SPEC CPU95.

6.3 Target Architecture & Parameters

We simulate four target architectures: the base processor, lockstepping, SRT, and CRT. We listed the basic processor's parameters in Table 1.

For the lockstepped processor, we chose two configurations. The first one—Lock0—has an ideal (but unrealistic) checker that executes in zero cycles. The second one—Lock8—is a more realistic checker that incurs an eight-cycle penalty to do the lockstep checking. We believe 8 cycles is realistic because of dense wiring to the centralized checker, wire delays, latency within

IEEE COMPUTER SOCIETY

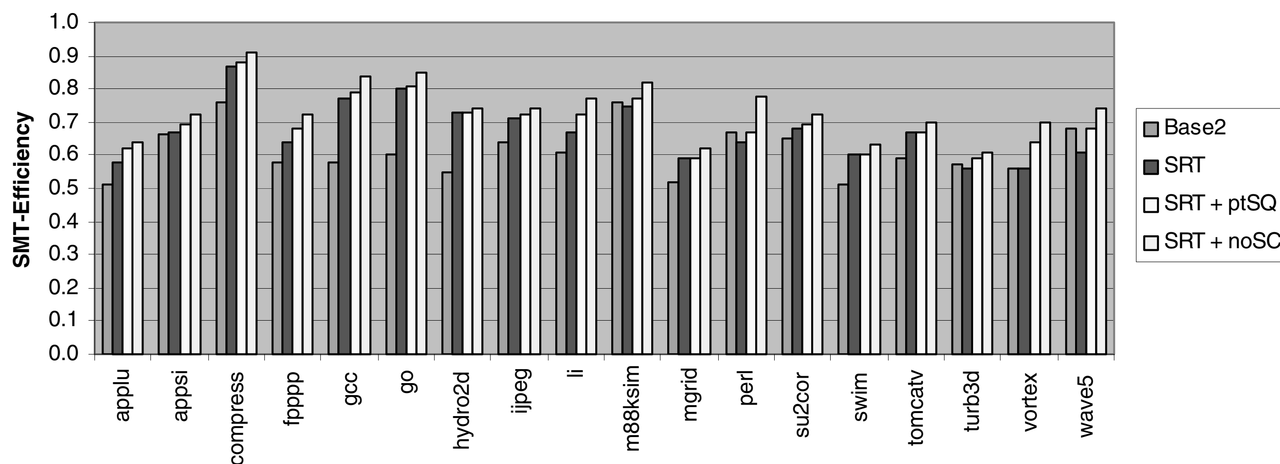


Figure 6. SMT-Efficiencies for one logical thread for four single-processor configurations. Base2 = base processor running one logical thread with two redundant copies, but no input replication or output comparison. SRT + ptSQ = SRT with per-thread store queue (with each store queue having 64 entries). SRT + noSC = SRT with no Store Comparison. "1.0" on the vertical axis represents the SMT-Efficiency of the base processor for one logical thread (running the same program) with no redundant copies.

the checker itself to do the comparisons, and the logic required to tolerate minor synchronization errors between processors. Many checkers allow processors to go out of lockstep briefly to tolerate non-deterministic timing, as can occur when one processor requires an extra cycle to address a correctable ECC fault in an internal structure such as the TLB.

For the SRT architecture, we conservatively assumed 4 cycles to forward line predictions from the QBOX to IBOX and 2 cycles to forward load values from the QBOX to the MBOX. For the CRT architecture, we assumed an additional 4-cycle penalty to forward data from one processor to the other processor's line prediction queue, load value queue, and store comparator.

6.4 Evaluation Metric

Instructions per cycle (IPC) is not a useful metric to compare SMT architectures, even though it has proven to be quite useful for single-threaded machines [22][19]. An SMT architecture may improve its overall IPC by favoring a more efficient thread (perhaps one with fewer cache misses and branch mispredictions), simply avoiding the challenging part of the workload.

Instead, we use *SMT-Efficiency* as a metric to evaluate the performance of our SRT, lockstepped, and CRT architectures. We compute SMT-Efficiency of an individual thread as the IPC of the thread in SMT mode (and running with other threads) divided by the IPC of the thread when it would run in single-thread mode through the same SMT machine. Then, we compute the SMT-Efficiency for all threads, as the arithmetic mean of the SMT-Efficiencies of individual threads. This arithmetic mean is the same as Snavely and Tullsen's weighted speedup metric [22].

7. RESULTS

This section evaluates the performance of the SRT, lockstepping, and CRT techniques using one or more independent logical threads. A logical thread runs an independent program, such as gcc. Normally, without any fault detection, a logical thread maps to a single hardware thread. However, in redundant mode, a logical thread is further decomposed into two hardware threads, each running a redundant copy of the program. Section 7.1 examines the performance of our SRT design using one and two logical threads. This section also shows how preferential space redundancy can improve SRT's fault coverage. Section 7.2 compares the performance of lockstepped and CRT processors using one, two, and four logical threads.

7.1 SRT

This section begins by analyzing the impact of preferential space redundancy, then evaluates the performance of our SRT design for one and two logical threads in detail.

7.1.1 Preferential Space Redundancy

This section examines the preferential space redundancy technique described in Section 4.5. We changed the scheduling policy of the SRT processor to direct corresponding instructions from the redundant threads to different halves of the QBOX, thereby improving fault coverage.

Figure 7 shows that, on average, without preferential space redundancy 65% of instructions go to the same functional unit. The fraction of corresponding instructions entering the same functional unit is higher than 50% because instructions are directed to a specific half of the queue based on their positions in the chunk forwarded to the QBOX. It is likely that instructions in both the leading and trailing threads will have instructions in similar positions of their chunks, which force them to go to the same half of the queue and, eventually, to the same functional unit. However, enabling preferential space redundancy reduces such instructions to 0.06%, thereby dramatically improving the fault coverage of the processor. The number is non-zero because if a different half is not available for the trailing thread, then the scheduler is forced to issue it in the same half. This technique, however, provides no performance degradation (not shown here), and in a few cases, such as hydro2d, improves performance because of better load balancing on the QBOX halves. Our remaining results in this section use preferential space redundancy.

7.1.2 One Logical Thread

Figure 6 shows the performance of the SRT processor for one logical thread. SRT, on average, degrades performance over running just the single thread (without any redundant copies) by 32%. However, SRT techniques improve performance over

IEEE COMPUTER SOCIETY

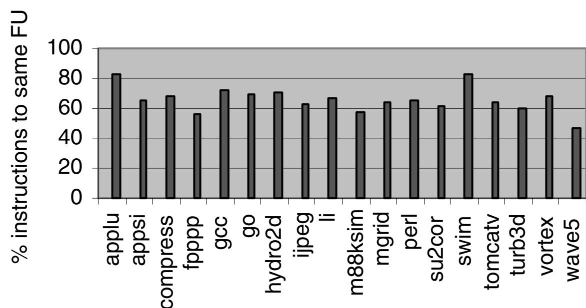


Figure 7. Percentage of corresponding instructions from redundant threads in the SRT processor entering the same functional unit (for one logical thread) in the absence of preferential space redundancy. With preferential space redundancy (not shown), the fraction of corresponding instructions entering the same functional unit is almost zero.

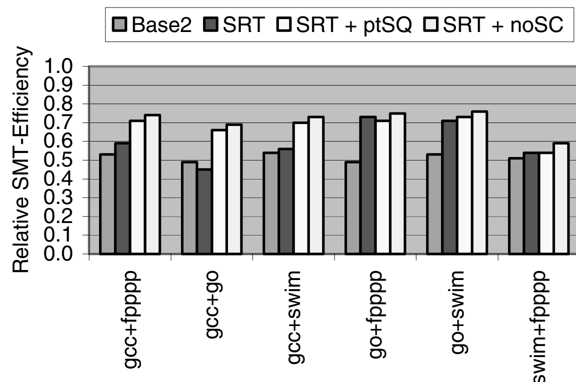


Figure 8. Relative SMT-Efficiencies for two logical threads for four single-processor configurations. The numbers are relative to SMT-Efficiencies for two logical threads running through our base processor without any redundancy. Base 2 = base processor running two logical threads, each with two redundant copies, but no input replication or output comparison. SRT + ptSQ = SRT with per-thread store queue (with each store queue having 64 entries). SRT + noSC = SRT with no store comparator.

running two redundant copies of the same program (without any input replication or output comparison)—base2 in the figure—by 11%. This improvement is due to the positive effects of the load value queue and line prediction queue in the SRT processor. The load value queue reduces data-cache misses in two ways: the trailing thread cannot miss, as it never directly accesses the cache, and the leading thread thrashes less in “hot” cache sets because it does not compete with the trailing thread. We find that the SRT processor, on average, has 68% fewer data cache misses compared to the base processor running redundant copies of two threads.

The store comparator is one of the key bottlenecks in the SRT design. As explained in Section 4.2, the store comparator increases the lifetime of a leading thread’s stores, which must now wait for the corresponding stores from the trailing thread to show up before they can retire. On average, for one logical thread, the store comparator increases the lifetime of a leading thread’s store by 39 cycles. Eighteen of these cycles represent the minimum latency for the trailing-thread store to fetch and execute; the extra 21 cycles come from queuing delays in the line prediction queue and processor pipeline.

Consequently, increasing the size of the store queue has significant impact on performance because this allows other stores from the leading thread to make progress. Using a per-thread store queue (with 64 entries per thread) improves the SMT-Efficiency by 4%, bringing the degradation to only roughly 30%. Completely eliminating the impact of the store comparator (SRT + noSC in the figure), perhaps with an even bigger store queue, would improve performance by another 5% and reduce the performance degradation to 26%.

Our 30% performance degradation for an SRT processor (with the per-thread store queue) is higher than our prior work [15], which reported only 21% degradation. We believe this discrepancy arises because our base processor’s structures are optimized primarily for uniprocessor performance, making its multithreaded performance relatively worse than that reported by our Simplescalar/SMT model.

7.1.3 Two Logical Threads

Interestingly, the per-thread store queue provides significantly greater benefits for two logical threads. Figure 8 shows results for the SRT processor variants for two logical threads. The overall degradation of the base SRT processor is 40% on average, about 8% higher than experienced by one logical thread, due to the greater resource pressures caused by the additional threads. However, adding a per-thread store queue significantly boosts the performance (by 15%) and reduces the degradation to only 32%, which is comparable to the 30% degradation experienced by one logical thread. The performance boost from the per-thread store queue is higher because the average lifetime of a leading thread’s store goes up to 44 cycles with two logical threads (compared to 39 cycles for one logical thread). Eliminating the store comparator entirely would provide only another 5% boost in performance, indicating that other resources are now the primary bottleneck.

The per-thread store queue provides significantly greater improvement in SMT-Efficiencies for gcc+fpppp, gcc+go, and gcc+swim compared to go+fpppp, go+swim, and swim+fpppp. Nevertheless, the overall IPC improvements of the SMT machines are comparable across all six benchmark configurations. The IPC improvements are 23%, 18%, 11%, 26%, 13%, and 9%, respectively for the six configurations. This suggests that the absence of the per-thread store queue penalizes the threads with low IPCs more than it penalizes threads with high IPCs, thereby decreasing the overall SMT-Efficiency.

7.2 CRT

This section compares the performance of our CRT processor with lockstepping for one, two, and four logical threads. We examine two versions of lockstepping—Lock0 and Lock8. Lock0 is an unrealistic implementation with a zero-cycle penalty for the checker. Lock8 is a more realistic implementation with an eight-cycle penalty for the checker.

7.2.1 One Logical Thread

Figure 9 compares the performance of CRT variants with lockstepping for one logical thread. For a single logical thread, a CRT processor performs similarly to Lock8 (about 2% better on average). This result is expected, because the CRT processor’s leading thread, which behaves similarly to the threads in the lockstepped processor, dominates its performance. The slight improvement in performance arises because all L2 cache misses incur higher penalty in Lock8 due to the presence of the checker. The absence of the checker (as in Lock0, the base case for the figure) would improve Lock8’s performance by about 5%.

IEEE COMPUTER SOCIETY

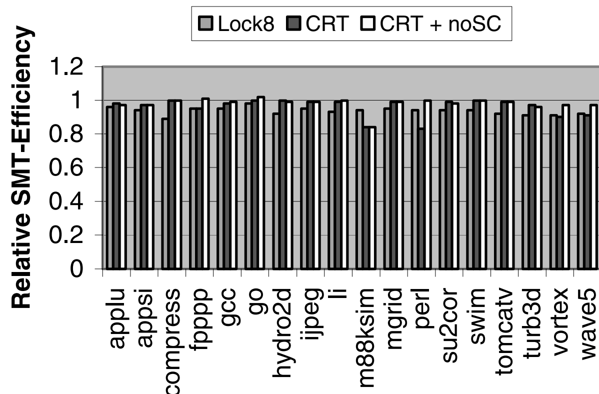


Figure 9. Comparison of Relative SMT-Efficiencies of Lockstepped and CRT processors for one logical thread. The numbers are relative to lockstepped processor with zero-cycle penalty for the checker. Lock8 is the lockstepped processor with an eight-cycle penalty for the checker. CRT is the Chip-Level Redundantly Threaded processor. CRT + noSC is the CRT processor with no store comparator (SC).

7.2.2 Two logical threads

Figure 10 compares CRT variants with lockstepping for two logical threads. On average, CRT outperforms Lock8 by 10% and Lock0 by about 2%. This performance improvement arises due to the cache and misspeculation effects of CRT. First, the presence of multiple threads creates cache contention in the lockstepped processors (where both threads contend in both caches) but not in the CRT processors (where the trailing threads get their data solely via the load value queue, leaving each leading thread with exclusive use of one cache). As a result, the CRT processor incurs 61% fewer data cache misses. In addition, the larger number of misses in the lockstepped configuration increases the performance impact of the checker penalty.

Second, CRT trailing threads do practically no misspeculation, unlike the lockstepped processors on which both threads misspeculate equally. The CRT processor has 24% fewer squashed instructions compared to a lockstepped processor. The percentage is lower than 50% because the CRT leading thread does more misspeculation than a lockstepped processor thread. Relative to a lockstepped thread, the CRT trailing thread uses reduced resources, allowing the CRT leading thread to run faster and, thereby, misspeculate more.

Adding the per-thread store queue to the CRT processor further improves the performance by 6%, making the CRT processor 13% better in performance than Lock8 on average, with a maximum improvement of 22%. The average lifetime of a store in a CRT processor goes up to 69 cycles (compared to 39 cycles for SRT and 49 cycles for CRT with one logical thread), so clearly a bigger store queue helps. Eliminating the store comparator completely, as in the CRT + noSC configuration, would give it another boost of 6%.

7.2.3 Four Logical Threads

Figure 11 compares the performance of CRT variants with lockstepping for four logical threads. Interestingly, unlike for two logical threads, CRT with a shared store queue performs similarly to Lock8. This result occurs because of the smaller number of entries per thread in the store queue. Thus, adding the per-thread store queue makes the CRT processor 13% better than Lock8 in performance on average, with a maximum improvement of 22%, which is similar to the improvements for two logical threads.

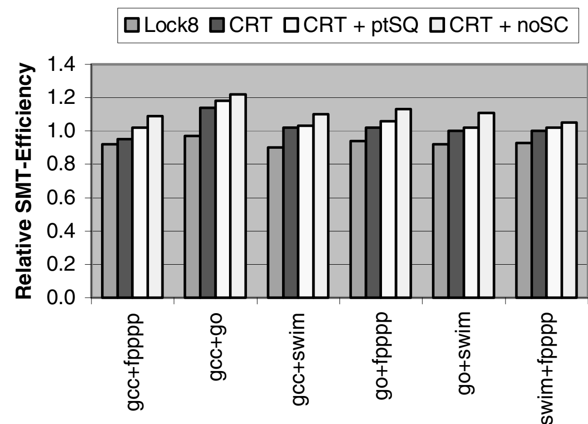


Figure 10. Comparison of Relative SMT-Efficiencies of Lockstepped and CRT processors for two logical threads. The numbers are relative to lockstepped processor with zero-cycle penalty for the checker. Lock8 is the lockstepped processor with 8 cycle penalty for the checker. CRT is the Chip-Level Redundantly Threaded processor. CRT + ptSQ =CRT with per-thread store queue. CRT + noSC is the CRT processor with no store comparator (SC).

Eliminating the store comparator, however, would only improve performance by another 2%.

8. RELATED WORK

Section 8.1 and 8.2 discuss related work in detecting faults in a single processor and dual processor systems, respectively. Section 8.3 discusses how our work relates to recent proposals for fault recovery.

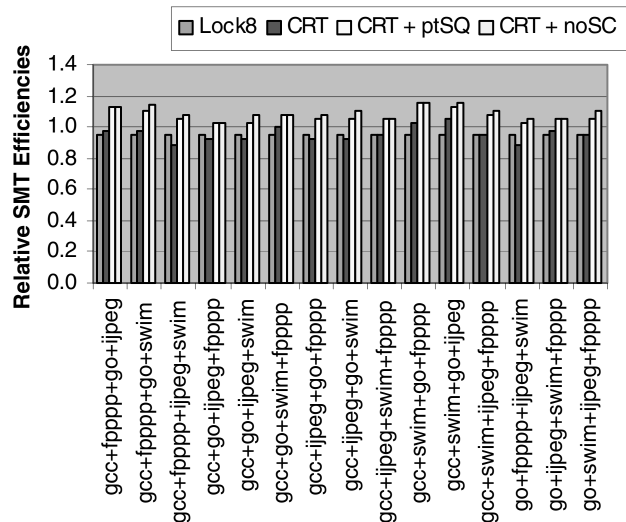


Figure 11. Comparison of Relative SMT-Efficiencies of Lockstepped and CRT processors for four logical threads. The numbers are relative to lockstepped processor with zero-cycle penalty for the checker. Lock8 is the lockstepped processor with 8 cycle penalty for the checker. CRT is the Chip-Level Redundantly Threaded processor. CRT + ptSQ =CRT with per-thread store queue. CRT + noSC is the CRT processor with no store comparator (SC).

IEEE COMPUTER SOCIETY

8.1 Fault Detection using a Single Processor

Several researchers have proposed the use of RMT techniques in single-processor SMT devices to detect transient faults. Saxena and McCluskey [18] were the first to propose the use of SMT processors for transient fault detection. Subsequently, Rotenberg's AR-SMT [17] and our SRT design [15] expanded this idea and proposed mechanisms to efficiently implement RMT techniques on a single-processor SMT device. We improve upon this body of prior work in four ways. First, we uncovered subtle implementation complexities by attempting to design a single-processor RMT processor based on a pre-existing SMT core resembling a commercial-grade microprocessor. For example, we had to replace the branch outcome queue—used both by AR-SMT and SRT—with a line prediction queue to help replicate the instruction stream in the redundant threads. Similarly, we found subtleties in the implementations of the load value replicator and store instruction comparator.

Second, we found that RMT techniques can incur a higher performance penalty for single-processor devices than previous studies on AR-SMT and SRT processors have demonstrated. Specifically, we find that the size of the store queue for the leading thread has a significant impact on the performance of single-processor RMT devices. This result led us to propose the use of per-thread store queues to enhance performance.

Third, we provide the first characterization of the performance of RMT devices with multithreaded workloads. We found that, with multithreaded workloads, the store queue size has a significantly higher impact on the performance of single-processor RMT devices. Nevertheless, with the use of per-thread store queues, the performance penalty of multithreaded workloads is similar to that of single-threaded workloads.

Finally, we demonstrated that even single-processor RMT devices could be modified effectively (e.g., with the use of preferential space redundancy) to significantly improve coverage of permanent faults, unlike prior work that only focused on transient fault detection for these devices.

Several researchers (e.g., [3,4,6,7,9,11]) have proposed a host of other non-multithreaded techniques for fault detection for uniprocessors.

8.2 Fault Detection using Two Processors

Lockstepped dual processors—both on a single die and different dies—have long been used for fault detection. Commercial fault-tolerant computers used for mission-critical applications have typically employed two processors with cycle-by-cycle lockstepping, such as the IBM S/390 with the G5 processor [21] and the Compaq Himalaya system [30].

Recently, Mahmood and McCluskey [9], Austin [1], and Sundaramoorthy, et al. [26] proposed the use of RMT techniques on dual-processor CMP cores. Mahmood and McCluskey's design uses a main processor core and a watchdog processor that compares its outputs with the outputs of the main processor. Austin's DIVA processor employs two processor cores—an aggressive high-performance processor, resembling a leading thread, and a low-performance checker processor, resembling a trailing thread. Because the processor cores are different, Austin's DIVA processor can potentially detect design faults, in addition to transient and permanent faults. Sundaramoorthy, et al.'s Slipstream processor uses a variant of AR-SMT on CMP processors. Although a Slipstream processor improves fault coverage, it cannot detect all single transient or permanent faults because it does not replicate all instructions from the instruction stream.

We improve upon this body of work on fault detection using dual processor cores in two ways. First, we show that the efficiency techniques of SRT can be extended to dual-processor CMPs. Second, we compared the performance of these CRT processors with on-chip lockstepping, using both single-threaded and multithreaded workloads. We demonstrated that CRT processors provide little advantage for single-threaded workloads, but perform significantly better than lockstepped processors for multithreaded workloads.

8.3 Fault Recovery

Recently, Vijaykumar et al. [29] proposed an architecture called SRTR, which extends SRT techniques to support transparent hardware recovery. SRTR compares instructions for faults before they retire and relies on the processor's intrinsic checkpointed state for recovery. Unlike SRTR, the RMT techniques in this paper assume that instructions are compared for faults after the instructions retire and rely on explicit software checkpoints (e.g., as in Tandem systems [30]) or hardware checkpoints (e.g., [25], [13]) for recovery.

9. CONCLUSIONS

Exponential growth in the number of on-chip transistors, coupled with reductions in voltage levels, has made microprocessors extremely vulnerable to transient faults. In a multithreaded environment, we can detect these faults by running two copies of the same program as separate threads, feeding them identical inputs, and comparing their outputs, a technique we call Redundant Multithreading (RMT).

This paper studied RMT techniques in the context of both single- and dual-processor simultaneous multithreaded (SMT) single-chip devices. Using a detailed, commercial-grade, SMT processor design we uncovered subtle RMT implementation complexities in the implementation of the load value queue, line prediction queue, and store comparator—structures that were necessary for efficient implementation of a single-processor RMT device.

We found that RMT techniques may have a more significant performance impact on single-processor devices than prior studies indicated. RMT degraded performance on single-threaded and multithreaded workloads on a single processor on average by 30% and 32%, respectively, noticeably higher than prior studies indicated. We also found that the store queue size could have a significant impact on performance. Because simply increasing the store queue size is likely to impact the processor cycle time, we proposed the use of per-thread store queues to allow greater number of store queue entries per thread.

We also demonstrated that a single-processor RMT device could not only cover transient faults, but could also significantly improve its permanent fault coverage by using a technique called preferential space redundancy. Preferential space redundancy directs a processor to choose space over time redundancy, given a choice between the two.

Although RMT techniques could be a significant performance burden for single-processor SMT devices, we found that a novel application of RMT techniques in a dual-processor device, which we term chip-level redundant threading (CRT), showed higher performance than lockstepping, especially on multithreaded workloads. We demonstrated that a CRT dual processor outperforms a pair of lockstepped CPUs by 13% on average (with a maximum improvement of 22%) on multithreaded workloads. This makes CRT a viable alternative for fault detection in upcoming dual-processor devices.

IEEE COMPUTER SOCIETY

Acknowledgments

We thank Bob Jardine and Alan Wood for inspiring many of the ideas in this paper. We thank George Chrysos, Joel Emer, Stephen Felix, Tryggve Fossum, Chris Gianos, Matthew Mattina, Partha Kundu, and Peter Soderquist for helping us understand the intricacies of the Alpha 21464 processor architecture. We also thank Eric Borch, Joel Emer, Artur Klauser, and Bobbie Manne for helping us with the Asim modeling environment. Finally, we thank Joel Emer and Geoff Lowney for providing helpful comments on initial drafts of this paper.

References

[1] Todd M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. 32nd Annual Int'l Symp. on Microarchitecture*, pp. 196-207, Nov. 1999.

[2] George Chrysos and Joel Emer, "Memory Dependence Prediction using Store Sets," *Proc. 25th Int'l Symp. on Computer Architecture*, pp. 142-153, Jun. 1998.

[3] Joel S. Emer, "Simultaneous Multithreading: Multiplying Alpha Performance," *Microprocessor Forum*, Oct. 1999.

[4] Joel Emer, Pritpal Ahuja, Nathan Binkert, Eric Borch, Roger Espasa, Toni Juan, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, and Steven Wallace, "Asim: A Performance Model Framework", *IEEE Computer*, 35(2):68-76, Feb. 2002.

[5] Manoj Franklin, "Incorporating Fault Tolerance in Superscalar Processors," *Proc. 3rd Int'l Conf. on High Performance Computing*, pp. 301-306, Dec. 1996.

[6] John G. Holm and Prithviraj Banerjee, "Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions," *Proc. Int'l Conf. on Parallel Processing*, Vol. I, pp. 192-195, Aug. 1992.

[7] IBM, "Power4 System Microarchitecture," http://www1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html.

[8] David J.C. Johnson, "HP's Mako Processor," Fort Collins Microprocessor Lab, Oct. 16, 2001. http://cpus.hp.com/technical_references/mpf_2001.pdf.

[9] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Trans. on Computers*, 37(2):160–174, Feb. 1988.

[10] Shubhendu S. Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb, "The 21364 Network Architecture," *IEEE Micro*, 22(1):26-35, Jan/Feb 2002.

[11] Janak H. Patel and Leona Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. on Computers*, 31(7):589–595, Jul. 1982.

[12] R. Preston, et al., "Design of an 8-Wide Superscalar RISC Microprocessor with Simultaneous Multithreading," *Digest of Technical Papers, 2002 IEEE International Solid State Circuits Conference*, pp. 334-335, San Francisco, CA.

[13] Milos Prvulovic, Zheng Zhang, and Josep Torrellas, "Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *Proc. 29th Annual Int'l Symp. on Computer Architecture*, May 2002.

[14] Joydeep Ray, James Hoe, and Babak Falsafi, "Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery," *Proc. 34th Int'l Symp. on Microarchitecture*, pp. 214-224, Dec. 2001.

[15] Steven K. Reinhardt and Shubhendu S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *Proc. 27th Int'l Symp. on Computer Architecture*, Jun. 2000.

[16] Dennis A. Reynolds and Gernot Metze, "Fault Detection Capabilities of Alternating Logic," *IEEE Trans. on Computers*, 27(12):1093–1098, Dec. 1978.

[17] Eric Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," *Proc. of Fault-Tolerant Computing Systems*, pp. 84-91, Jun. 1999.

[18] N. R. Saxena and E. J. McCluskey, "Dependable Adaptive Computing Systems," *Proc. IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, Oct. 11-14, 1998.

[19] Yiannakis Sazeides and Toni Juan, "How to Compare the Performance of Two SMT Architectures," *Proc. 2001 Int'l Symp. on Performance Analysis of Systems and Software*, Nov. 2001.

[20] Daniel P. Siewiorek and Robert S. Swarz, *Reliable Computer Systems: Design and Evaluation*, A.K. Peters Ltd, Oct. 1998.

[21] T. J. Slegel, et al., "IBM's S/390 G5 Microprocessor Design," *IEEE Micro*, 19(2):12–23, Mar/Apr 1999.

[22] Allan Snavely and Dean Tullsen, "Symbiotic Job scheduling for a Simultaneous Multithreading Processor," *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 234–244, Nov. 2000.

[23] G. S. Sohi, M. Franklin, and K. K. Saluja, "A Study of Time-Redundant Fault Tolerance Techniques for High-Performance Pipelined Computers," *Digest of Papers, 19th Int'l Symp. on Fault-Tolerant Computing*, pp. 436–443, 1989.

[24] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. on Computers*, 39(3):349–359, March 1990.

[25] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," *Proc. 29th Annual Int'l Symp. on Computer Architecture*, May 2002.

[26] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance," *Proc. Ninth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 257–268, Nov. 20001.

[27] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Annual Int'l Symp. on Computer Architecture*, pp. 392-403, Jun. 1995.

[28] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Annual Int'l Symp. on Computer Architecture*, pp. 191-202, May 1996.

[29] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng, "Transient Fault Recovery using Simultaneous Multithreading," *Proc. 29th Annual Int'l Symp. on Computer Architecture*, May 2002.

[30] Alan Wood, "Data Integrity Concepts, Features, and Technology," White paper, Tandem Division, Compaq Computer Corporation.

[31] Wayne Yamamoto and Mario Nemirovsky, "Increasing Superscalar Performance Through Multistreaming," *Proc. 1995 Annual Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 49–58, June 1995.