Transient Fault Detection via Simultaneous Multithreading

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

Shubhendu S. Mukherjee
VSSAD, Alpha Technology Group
Compaq Computer Corporation
334 South Street, Mail Stop SHR3-2E/R28
Shrewsbury, MA 01545
Shubhendu.Mukherjee@compaq.com

ABSTRACT

Smaller feature sizes, reduced voltage levels, higher transistor counts, and reduced noise margins make future generations of microprocessors increasingly prone to transient hardware faults. Most commercial fault-tolerant computers use fully replicated hardware components to detect microprocessor faults. The components are lockstepped (cycle-by-cycle synchronized) to ensure that, in each cycle, they perform the same operation on the same inputs, producing the same outputs in the absence of faults. Unfortunately, for a given hardware budget, full replication reduces performance by statically partitioning resources among redundant operations.

We demonstrate that a Simultaneous and Redundantly Threaded (SRT) processor—derived from a Simultaneous Multithreaded (SMT) processor—provides transient fault coverage with significantly higher performance. An SRT processor provides transient fault coverage by running identical copies of the same program simultaneously as independent threads. An SRT processor provides higher performance because it dynamically schedules its hardware resources among the redundant copies. However, dynamic scheduling makes it difficult to implement lockstepping, because corresponding instructions from redundant threads may not execute in the same cycle or in the same order.

This paper makes four contributions to the design of SRT processors. First, we introduce the concept of the sphere of replication, *which abstracts both the physical redundancy of a lockstepped system and the logical redundancy of an SRT processor. This framework aids in identifying the scope of fault coverage and the input and output values requiring special handling. Second, we identify two viable spheres of replication in an SRT processor, and show that one of them provides fault detection while checking only committed stores and uncached loads. Third, we identify the need for consistent replication of load values, and propose and evaluate two new mechanisms for satisfying this requirement. Finally, we propose and evaluate two mechanisms—slack fetch and branch outcome queue—that enhance the performance of an SRT processor by allowing one thread to prefetch cache misses and branch results for the other thread. Our results with 11 SPEC95 benchmarks show that an SRT processor can outperform an equivalently sized, on-chip, hardware-replicated solution by 16% on average, with a maximum benefit of up to 29%.*

Steven K. Reinhardt is supported by a National Science Foundation CAREER award (CCR-9734026), a grant from Compaq, and gifts from Intel and IBM.

This paper appeared in the 27th Annual International Symposium on Computer Architecture, June 2000.

1. INTRODUCTION

Modern microprocessors are susceptible to transient hardware faults. For example, cosmic rays can alter the voltage levels that represent data values in microprocessor chips. The energy flux of these rays can be reduced to acceptable levels with six feet or more of concrete. Unfortunately, this width is significantly greater than normal computer room roofs or walls. Currently, the frequency of such transient faults is low—typically less than one fault per year per thousand computers [20]—making fault-tolerant computers attractive only for mission-critical applications, such as online transaction processing and space programs. However, future microprocessors will be more prone to transient faults due to their smaller feature sizes, reduced voltage levels, higher transistor counts, and reduced noise margins. Hence, in the future, even low-end personal computing systems may need support for fault detection, if not for recovery.

Current fault-tolerant commercial systems, such as the Compaq NonStop Himalaya [20] and the IBM S/390 [13] systems, detect hardware faults using a combination of space redundancy (execution on replicated hardware) and information redundancy (e.g., ECC or parity). For example, Compaq Himalaya systems detect microprocessor faults by running identical copies of the same program on two identical *lockstepped* (cycle-synchronized) microprocessors. In each cycle, both processors are fed identical inputs, and a checker circuit compares their outputs. On an output mismatch, the checker flags an error and initiates a software recovery sequence.

In this paper, we investigate the use of simultaneous multithreading as a hardware mechanism to detect transient hardware faults. Simultaneous multithreading (SMT) [18][19][21] is a novel technique to improve the performance of a superscalar microprocessor. An SMT machine allows multiple independent threads to execute simultaneously—that is, in the same cycle—in different functional units. For example, the Alpha 21464 [3] implements a four-threaded SMT machine that can issue up to eight instructions per cycle from one or more threads.

As noted previously by Rotenberg [11], an SMT processor is attractive for fault detection because it can provide redundancy by running two copies of the same program simultaneously. Such a simultaneous and redundantly threaded (SRT) processor provides three advantages over conventional hardware replication. First, an SRT processor may require less hardware because it can use time

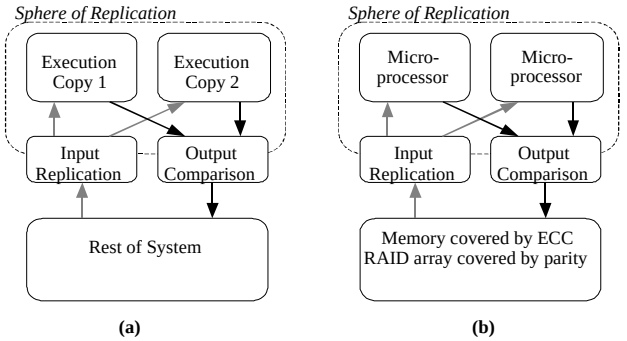

Figure 1. Sphere of Replication (a) General (b) Compaq NonStop Himalaya system.

and information redundancy in places where space redundancy is not critical. For example, an SRT processor can have an unreplicated datapath protected by ECC or parity and shared between redundant threads. Second, an SRT processor can potentially provide performance superior to an equivalently sized hardware-replicated solution because it partitions resources among execution copies dynamically rather than statically. Third, SRT processors may be more economically viable, since it should be feasible to design a single processor that can be configured in either SRT or SMT mode depending on the target system. This merged design would enjoy the combined market volume of both high-reliability and high-throughput processors.

Unfortunately, an SRT processor poses two critical design challenges. First, lockstepping—that is, cycle-by-cycle output comparison and input replication—is inappropriate for a dynamically-scheduled SRT processor. An instruction in one thread is likely to execute in a different cycle than its equivalent in the other thread, and in a different order with respect to other instructions from the same thread. Due to speculation, one thread may even execute instructions that have no equivalent in the other thread. Consequently, more sophisticated techniques are required to compare outputs and replicate inputs.

Second, redundant thread interactions in an SRT processor must be orchestrated to provide peak efficiency. For example, if both threads encounter a cache miss or branch misprediction simultaneously, both threads will stall or misspeculate at the same time, reducing the benefit of dynamic resource sharing.

This paper makes four contributions in identifying and evaluating key mechanisms to solve the above problems.

First, we introduce the concept of the *sphere of replication* (Figure 1), the logical domain of redundant execution. The sphere of replication abstracts both the physical redundancy of a lockstepped system and the logical redundancy of an SRT processor. Components inside the sphere of replication enjoy fault coverage due to redundant execution, while components outside the sphere do not (and hence must be covered by other techniques, such as ECC). Values that enter and exit the sphere are the inputs and outputs that require replication and comparison, respectively. Correctly identifying a system's sphere of replication aids in determining a set of replication and comparison mechanisms that are sufficient but not superfluous. By varying the extent of the sphere of replication, an SRT processor designer can influence this set, affecting the complexity and cost of the required hardware.

Our second contribution applies the sphere of replication framework to identify two viable design alternatives for output comparison in SRT processors: checking memory accesses only (stores and uncached loads), or checking memory accesses and register updates. Our results indicate that either technique can be implemented with almost no performance penalty, but checking memory accesses only incurs less hardware overhead.

Our third contribution is to identify the need for consistent replication of memory inputs (load values), and to propose and evaluate two new mechanisms for load value replication in SRT processors, namely the *Active Load Address Buffer (ALAB)* and the *Load Value Queue (LVQ)*. The ALAB allows corresponding cached loads from both replicated threads to receive the same value in the presence of out-of-order and speculative execution of either thread, cache replacements, and multiprocessor cache invalidations. In contrast, the LVQ uses a single cache access to satisfy both threads, forwarding the (pre-designated) leading thread's committed load addresses and values to the trailing thread. The trailing thread derives all its load values from the LVQ instead of the data cache. Our results show that both techniques incur very little hardware overhead and almost no performance penalty, but the LVQ is a much simpler design.

Our fourth contribution is to propose and evaluate two techniques that enhance performance in an SRT processor: *slack fetch* and *branch outcome queue*. Slack fetch is a new mechanism that tries to maintain a constant slack of instructions between the two replicated threads. This slack allows one thread to "warm up" the caches and branch predictor for the other thread, thereby improving the trailing thread's performance. The branch outcome queue improves upon the slack fetch mechanism by forwarding correct branch outcomes (branch target addresses) from one thread to the other. The trailing thread executes a branch only when its outcome is available from the branch outcome queue. Consequently, the trailing thread never misspeculates on branches.

Overall, our results—with 11 benchmarks from the SPEC95 suite—show that an SRT processor can outperform an equivalently sized on-chip hardware-replicated solution by 16% on average, with a maximum benefit of up to 29%. An SRT processor shows superior performance due to the combination of SMT's dynamic resource scheduling and our performance-enhancing techniques (slack fetch and branch outcome queue). Furthermore, the performance impact of our output comparison and input replication techniques is negligible.

The rest of the paper is organized as follows. Section 2.1 discusses current solutions for transient fault detection. Section 2.2 describes the basics of Simultaneous Multithreading. Section 3 describes the design space of SRT processors and our proposals for output comparison, input replication, and performance enhancement. Section 4 describes our evaluation methodology and Section 5 discusses our results. Section 6 discusses further techniques to improve fault coverage in an SRT processor. Section 7 discusses related work, including the AR-SMT design [11], an independently developed example of an SRT processor. Finally, Section 8 presents our conclusions.

2. BACKGROUND

This section provides background on the two areas we combine in this paper: transient fault detection and simultaneous multithreading.

2.1 Hardware Detection of Transient Faults

Fault tolerance requires at least two basic mechanisms: fault detection and recovery. Fault detection enables the construction of fail-stop components: components that, in the case of failure, halt before they propagate erroneous outputs. Given fail-stop components, designers can employ several well-known recovery

techniques such as checkpoint/restart or failover to construct highly reliable systems [12]. We focus on fault detection mechanisms; fault recovery schemes are beyond the scope of this paper. More specifically, we concentrate on mechanisms to detect *transient* faults, which persist for only a short duration. Complete coverage for permanent faults often requires careful analysis of a specific processor implementation. Nevertheless, Section 6 discusses extensions to SRT processors that increase the coverage of permanent faults through redundant execution.

Most fault-tolerant computers, such as the IBM S/390 G5 [13] and NonStop Himalaya systems [12], detect hardware faults in hardware (as opposed to software). Fault detection in hardware is critical for two reasons. First, hardware fault detection allows a fault-tolerant computer to catch faults rapidly; in contrast, software fault detection typically incurs higher latency from fault to detection, which results in a much larger window of vulnerability in which an error can propagate to other components in a system. Second, hardware fault detection usually incurs lower performance overhead, resulting in less performance degradation due to the fault detection mechanism.

Hardware fault detection usually involves a combination of information redundancy (e.g., ECC, parity, etc.), time redundancy (e.g., executing the same instruction twice in the same functional unit, but at different times), and space redundancy (e.g., executing the same instruction in different functional units). Information redundancy techniques can efficiently detect faults on data that is stored or transmitted, such as in memory or on busses or networks. However, information redundancy techniques for covering control logic and computation units, such as self-checking circuits [12], may be more difficult to design. Time redundancy can more effectively detect faults in computation or control logic, but with a potentially high performance overhead (theoretically, up to 100%). As a result, space redundancy has become a common performance optimization for hardware fault detection.

Both the IBM S/390 G5 and Compaq NonStop Himalaya systems use space-redundant hardware to optimize performance, except where the cost of space redundancy is high. The IBM G5 microprocessor replicates the fetch, decode, and execution units of the pipeline on a single chip. In contrast, the Compaq Himalaya system achieves space redundancy using a pair of off-the-shelf microprocessors (e.g., the MIPS R10000). Both systems detect faults using lockstepping: in each cycle, the replicated components receive the same inputs, and their outputs are compared using custom circuitry. A mismatch implies that one of the components encountered a fault, and both components are halted before the offending output can propagate to the rest of the system. To reduce hardware cost, however, neither system replicates large components such as main memory; instead, both use ECC to cover memory faults.

2.2 Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT) [18][19][21] is a technique that allows fine-grained resource sharing among multiple threads in a dynamically scheduled superscalar processor. An SMT processor extends a standard superscalar pipeline to execute instructions from multiple threads, possibly in the same cycle.

Although our SRT techniques could apply to any SMT processor, we will use the CPU model shown in Figure 2 for illustration and evaluation. Our base processor design is inherited from the SimpleScalar "sim-outorder" code [2], from which our simulator is derived. In our SMT version of this machine, the fetch stage feeds instructions from multiple threads (one thread per cycle) to a simple fetch/decode queue. The decode stage picks

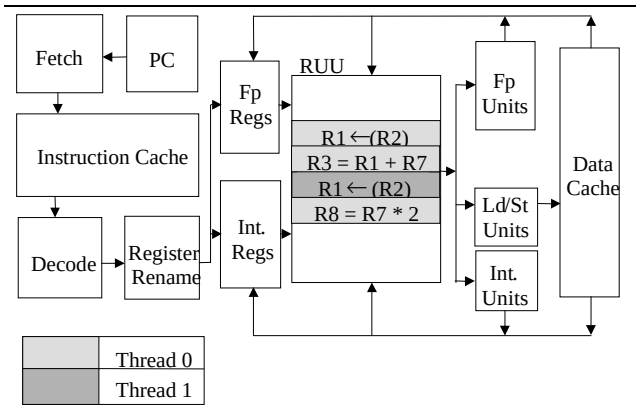

Figure 2. Sharing of RUU between two threads in our SMT processor model.

instructions from this queue, decodes them, locates their source operands, and places them into the Register Update Unit (RUU) [16]. The RUU serves as a combination of global reservation station pool, rename register file, and reorder buffer. Loads and stores are broken into an address generation and a memory reference. The address generation portion is placed in the RUU, while the memory reference portion is placed into a similar structure, the Load/Store Queue (LSQ) (not shown in Figure 2).

Figure 2 shows instructions from two threads sharing the RUU. Multiple instructions per cycle are issued from the RUU to the execution units and written back to the RUU without regard to thread identity. The processor provides precise exceptions by committing results for each thread from the RUU to the register files in program order. Tullsen, et al. [19] showed that optimizing the fetch policy—the policy that determines which thread to fetch instructions from in each cycle—can improve the performance of an SMT processor. The best-performing policy they examined was named ICount. The ICount policy counts the number of instructions from active threads that are currently in the instruction buffers and fetches instructions from the thread that has the fewest instructions. The assumption is that this thread is moving instructions through the processor quickly, and hence, making the most efficient use of the pipeline. Our base SMT machine uses the ICount policy. In Section 3.4, we will show how to modify this fetch policy to improve the performance of an SRT processor.

3. SRT DESIGN SPACE

We modify an SMT processor to detect faults by executing two redundant copies of each thread in separate thread contexts. Unlike true SMT threads, each redundant thread pair appears to the operating system as a single thread. All replication and checking are performed transparently in hardware. In this paper, we focus on extending an SMT processor with two thread contexts to support a single-visible-thread SRT-only device. Nevertheless, we can easily extend our design to support two OS-visible threads on an SMT machine with four thread contexts.

This section explores the SRT design space. Section 3.1 describes the sphere of replication concept, which helps to identify the scope of fault coverage in an SRT design and the output and input values requiring comparison and replication. This section also identifies two spheres of replication for SRT processors that we will analyze in the rest of this paper. Section 3.2 describes output comparison techniques, while section 3.3 describes input replication techniques. Finally, Section 3.4 describes two techniques to improve performance in an SRT processor.

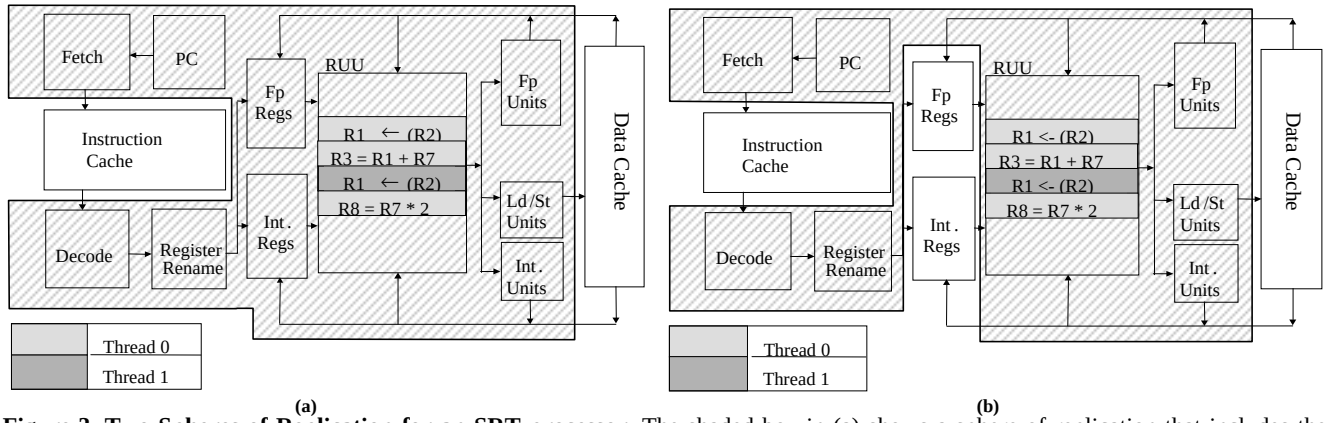

Figure 3. Two Spheres of Replication for an SRT processor. The shaded box in (a) shows a sphere of replication that includes the entire SMT pipeline shown in Figure 2, except the first level data and instruction caches. The shaded box in (b) shows a sphere of replication that excludes the architectural register file, the first-level data cache, and the first-level instruction cache.

3.1 Sphere of Replication

Three key design questions for SRT processors are:

- *For which components will the redundant execution mechanism detect faults?* Components not covered by redundant execution must employ alternative techniques, such as ECC.
- *Which outputs must be compared?* Failing to compare critical values compromises fault coverage. On the other hand, needless comparisons increase overhead and complexity without improving coverage.
- *Which inputs must be replicated?* Failure to correctly replicate inputs can result in the redundant threads following divergent execution paths.

In a lockstepped, physically-replicated design, answers to these questions are clear. Redundant execution captures faults within the physically replicated components. Outputs and inputs requiring comparison and replication are the values on the physical wires leading into and out of these components.

The answers to these questions are less obvious in an SRT design. To aid our analysis, we introduce the concept of the *sphere of replication* (Figure 1a), the logical extent of redundant execution. The sphere of replication abstracts both the physical redundancy of a lockstepped system and the logical redundancy of an SRT processor. All activity and state within the sphere is replicated, either in time or in space. Values that cross the boundary of the sphere of replication are the outputs and inputs that require comparison and replication, respectively.

Figure 1b shows how the sphere of replication applies to a Compaq Himalaya system. The entire microprocessor is within the sphere of replication and is space-replicated, i.e., there are two physical copies. Output comparison is performed by dedicated hardware that compares the signals on corresponding pairs of output pins in every cycle. Input replication is achieved simply by sending the same signals to corresponding pairs of input pins. As indicated in Figure 1b, system components outside the sphere rely on other techniques for fault coverage. The use of other fault detection techniques inside the sphere—e.g., ECC or parity in on-chip caches—is not strictly necessary.

In the IBM G5, the sphere of replication comprises of fetch, decode, and execution units. Output comparison involves checking register writeback values and memory store values. Because the register file is outside the sphere, output comparison must be done before values reach the registers, and the register file cannot rely on replication for fault coverage (the G5 uses ECC). Input replication consists of sending the same instructions and register values to both datapaths

As these two examples demonstrate, the extent of the sphere of replication impacts a number of system parameters. A larger sphere typically replicates more state (e.g., register file and cache contents). However, moving state into the sphere means that updates to that state occur independently in each execution copy. Thus, a larger sphere tends to decrease the bandwidth required for output comparison and input replication, potentially simplifying the comparator and replicator circuits. .

The sphere of replication in an SRT processor is physically less distinct because replication occurs through time redundancy as well as space redundancy. For example, the corresponding instructions from redundant threads may occupy the same RUU slot in different cycles (time redundancy), different RUU slots in the same cycle (space redundancy), or different slots in different cycles (both).

In this paper, we evaluate the two spheres of replication shown in Figure 3. Given the size and fraction of die area devoted to on-chip caches, and the ease with which these can be protected using parity or ECC, we assume that it is not effective to replicate them. Our first sphere of replication (Figure 3a) comprises the entire processor excluding caches. Our second sphere of replication (Figure 3b) is similar, but also excludes the architectural register file (as in the G5). Sections 3.2 and 3.3 describe how we perform output comparison and input replication for these spheres.

3.2 Output Comparison in an SRT Processor

Like conventional fault-tolerant systems, SRT processors detect faults by comparing outputs of redundant executions. The sphere of replication determines which values need to be compared. When the register file lies inside the sphere (Figure 3a), there are three types of values that exit the sphere:

- *Stores*. The checker must verify the address and data of every committed store before it forwards them outside the sphere of replication. We use an ordered, non-coalescing store buffer shared between the redundant threads to synchronize and verify store values as they retire in program order from the RUU/LSQ. Each thread has an independent tail pointer into the buffer. If a thread finds its tail entry uninitialized, it writes the address and data value of its store into the entry. The second thread will find this entry initialized, so it will compare its address and data with the existing values. On a match, the entry is marked as verified and issued to the data cache. A mismatch indicates a fault.

Misspeculated stores never send their data outside the sphere of replication, so they do not need checking. To provide each thread with a consistent view of memory, the store buffer forwards data to subsequent loads only if the store has retired in the thread issuing the load.

- *Cached load addresses*. Although cached data and instruction fetch addresses leave the sphere of execution, they do not affect the architectural state of the machine, so they do not require checking. If a faulty address leads to an incorrect load value or instruction, any resulting error will be detected via other output comparison checks before affecting architectural state outside the sphere. Section 5.2 will show that allowing one thread to issue cache fetches early (and without output comparison), effectively prefetching for the other thread, is critical to efficiency in SRT processors.
- *Uncached load addresses*. Unlike cached loads, uncached loads typically have side effects in I/O devices outside the sphere of replication, so these addresses must be checked. However, unlike stores, uncached load addresses must be compared before the load commits. Fortunately, in most processors, uncached loads issue non-speculatively, and only after all prior loads and stores commit. Also, no load or store after the uncached load in program order can issue until the uncached load commits. Thus an uncached load can simply stall in the execute stage until the corresponding instruction for the other thread arrives, at which point the addresses can be compared.

An undetected fault could occur if an uncached load address was erroneously classified as cached and allowed to proceed without being checked. We assume that adequate precautions are taken to prevent this specific case, such as additional physical-address cacheability checks. The detailed design of these checks is implementation-dependent.

The second sphere of replication (Figure 3b) does not contain the register file, so it requires output comparison on values sent to the register file—i.e., on register writebacks of committed instructions. As with stores, both the address (register index) and value must be verified.

Register writeback comparison could be done as instructions retire from the RUU. However, forcing every instruction to wait for its equivalent from the other thread significantly increases RUU occupancy. Since the RUU is a precious resource, we use a *register check buffer*, similar to the store buffer, to hold results from retired but unmatched instructions. The first instance of an instruction records its result in the buffer. When the corresponding instruction from the other thread leaves the RUU, the index and value are compared and, if they match, the register file is updated.

As with the store buffer, results in the register check buffer must be forwarded to subsequent instructions in the same thread to provide a consistent view of the register file. We can avoid complex forwarding logic by using the separate per-thread register files of the SMT architecture as "future files" [14]. That is, as each instruction retires from the RUU, it updates the appropriate per-thread register file, as in a standard SMT processor. This register file then reflects the up-to-date but unverified register contents for that redundant thread. As register updates are verified and removed from the register check buffer, they are sent to a third register file, which holds the protected, verified architectural state for the user-visible thread.

Having a protected copy of the architectural register file outside the sphere of replication simplifies fault recovery on an output mismatch, as the program can be restarted from the known good contents of the register file (as in the IBM G5 microprocessor [13]). However, this benefit requires the additional costs of verifying register updates and protecting the register file with ECC or similar coverage. Although the register check buffer is conceptually similar to our store buffer, it must be significantly larger (see Section 5.3) and must sustain higher bandwidth in updates per cycle to avoid degrading performance. Our larger sphere of replication shows that fault detection can be achieved without these costs.

3.3 Input Replication in an SRT Processor

Inputs to the sphere of replication must be handled carefully to guarantee that both execution copies follow precisely the same path. Specifically, corresponding operations that input data from outside the sphere must return the same data values in both redundant threads. Otherwise, the threads may follow divergent execution paths, leading to differing outputs that will be detected and handled as if a hardware fault occurred.[1]

As with output comparison, the sphere of replication identifies values that must be considered for input replication: those that cross the boundary into the sphere. For the first sphere of replication (Figure 3a), four kinds of inputs enter the sphere:

- *Instructions.* We assume that the contents of the instruction space do not vary with time, so that unsynchronized accesses from redundant threads to the same instruction address will return the same instruction without additional mechanisms. Updates to the instruction space require thread synchronization, but these updates already involve system-specific operations to maintain instruction-cache consistency in current CPUs. We believe these operations can be extended to enforce a consistent view of the instruction space across redundant threads.
- *Cached load data.* Corresponding cached loads from replicated threads must return the same value to each thread. Unlike instructions, data values may be updated by other processors or by DMA I/O devices between load accesses. To make matters worse, an out-of-order SRT processor may issue corresponding loads from different threads in a different order and in different cycles. Because of speculation, the threads may even issue different numbers of loads. In Section 3.3.1 we introduce two new mechanisms—*Active Load Address Buffer* and *Load Value Queue*—for input replication of cached load data.
- *Uncached load data.* As with cached load data, corresponding loads must return the same value to both threads. Because corresponding uncached loads must synchronize to compare addresses before being issued outside the sphere of replication, it is straightforward to maintain synchronization until the load data returns, then replicate that value for both threads. Other instructions that access non-replicated, time-varying state, such as the Alpha *rpcc* instruction that reads the cycle counter, are handled similarly.
- *External interrupts*. Interrupts must be delivered to both threads at precisely the same point in their execution. We envision two potential solutions. The first solution forces the threads to the same execution point by stalling the leading thread until the trailing thread catches up, then delivers the interrupt synchronously to both threads. (If the register file is outside the sphere of replication, then we could alternatively roll both threads back to the point of the last committed register write.)

[1] If fault recovery is rapid (e.g., done in hardware) and input replication failures are rare, these false faults may not affect performance significantly. However, if input replication is performed correctly, the need for rapid fault recovery is reduced.

The second solution delivers the interrupt to the leading thread, records the execution point at which it is delivered (e.g., in committed instructions since the last context switch), then delivers the interrupt to the trailing thread when it reaches the identical execution point.

As with output comparison, moving the register file outside the sphere means that additional values cross the sphere boundary. In the case of input replication, it is the register read values that require further consideration. However, each thread's register read values are produced by its own register writes, so corresponding instructions will receive the same source register values in both threads in the absence of faults (and assuming that all other inputs are replicated correctly). In fact, many source register values are obtained not from the register file, but by forwarding the results of earlier uncommitted instructions from the RUU (or from the "future file" as discussed in the previous section). Thus input replication of register values requires no special mechanisms even when the register file is outside the sphere of replication.

3.3.1 Input Replication of Cached Load Data

Input replication of cached load data is problematic because data values can be modified from outside the processor. For example, consider a program waiting in a spin loop on a cached synchronization flag to be updated by another processor. The program may count the number of loop iterations in order to profile wait times or adaptively switch synchronization algorithms. To prevent redundant threads from diverging, both threads must spin for an identical number of iterations. That is, the update of the flag must appear to occur in the same loop iteration in each thread, even if these corresponding iterations are widely separated in time. Simply invalidating or updating the cache will likely cause the leading thread to execute more loop iterations than the trailing thread.

We propose two mechanisms to input replication of cached load data: the Active Load Address Buffer (ALAB) and the Load Value Queue (LVQ). We describe these in the subsections below.

3.3.1.1 Active Load Address Buffer (ALAB)

The ALAB provides correct input replication of cached load data by guaranteeing that corresponding loads from redundant threads will return the same value from the data cache. To provide this guarantee, the ALAB delays a cache block's replacement or invalidation after the execution of a load in the leading thread until the retirement of the corresponding load in the trailing thread.

The ALAB itself comprises a collection of identical entries, each containing an address tag, a counter, and a pending-invalidate bit. When a leading thread's load executes, the ALAB is searched for an entry whose tag matches the load's effective address; if none is found, a new entry is allocated.[2] Finally, the entry's counter is incremented to indicate an outstanding load to the block. When a trailing thread's load retires, the ALAB is again searched and the matching entry's counter is decremented.[3]

When a cache block is about to be replaced or invalidated, the ALAB is searched for an entry matching the block's address. If a matching entry with a non-zero count is found, the block cannot be replaced or invalidated until all of the trailing thread's outstanding accesses to the block have completed. At this point, the counter will be zero and the block can be released. (An invalidation request may be acknowledged immediately; however, depending on the memory consistency model, the processor may have to stall other memory requests until the ALAB counter reaches zero.) To guarantee that the counter eventually reaches zero, the cache sets the ALAB entry's pending-invalidate bit to indicate that it is waiting; leading thread loads that attempt to increment an entry with a set pending-invalidate bit are stalled. Because the trailing thread can always make forward progress, the outstanding loads will eventually complete.

Because the ALAB must reliably track every outstanding load, non-zero ALAB entries cannot be evicted. A leading-thread load must stall if it cannot allocate a new entry due to ALAB mapping conflicts, or if it would increment the entry's counter beyond the maximum value. Because loads are executed out of order, it is possible that a leading-thread load will be forced to stall because loads that follow it in program order have saturated the counter or acquired all the conflicting entries in an ALAB set. The processor can detect this deadlock condition by recognizing when (1) the trailing thread has caught up to the leading thread and (2) the leading thread's oldest instruction is a load stalled because it cannot perform its ALAB increment. In this case, the processor must flush the leading thread from the pipeline (as on a misspeculation), decrementing the ALAB counters corresponding to executed but squashed loads. The stalled load can then reissue; because the trailing thread has committed all outstanding loads, it is guaranteed not to stall on the ALAB increment. Our results show that a 64-entry fully associative ALAB with 7-bit counters avoids this deadlock entirely across all of our benchmarks, while a 32-entry buffer encounters it rarely (thirty-five times out of the 2.2 billion instructions simulated across our eleven benchmarks).

3.3.1.2 Load Value Queue (LVQ)

The Load Value Queue (LVQ) is an alternate mechanism to implement input replication of cached load values. Compared to the ALAB, the LVQ is much simpler to implement. In this mechanism, we explicitly designate one of the redundant threads as a leading thread and the other one as a trailing thread. Only the leading thread issues cached loads to the data cache outside the sphere of replication. When a cached load commits, the leading thread sends the load's address and value to the trailing thread via the Load Value Queue (LVQ), which is a first-in first-out buffer protected by ECC. The trailing thread performs loads in program order and non-speculatively. The loads in the trailing thread proceed through the regular processor pipeline. However, instead of probing the data cache, the trailing thread waits for its corresponding load address and value from the leading thread to show up at the head of the LVQ. Input replication is guaranteed because the trailing thread receives the same value for the load that the leading thread used. Our results show that the LVQ performs input replication of cached load data with almost no performance penalty.

---

[2] The ALAB may be direct-mapped, set-associative, or fully associative.

[3] In the absence of address corruption, the trailing thread is guaranteed to find a matching entry because non-zero ALAB entries cannot be replaced. A corrupted address will be detected as the absence of a match at this point or as a deadlock condition later (if a spurious match occurs and the wrong entry is incremented/decremented). In any case, the fault will be caught by an output comparison mechanism before it propagates outside the sphere of replication.

Table 1. Simulated Processor Configuration.

L1 Instruction Cache	64K bytes, 4-way associative, 32-byte blocks, single ported
L1 Data Cache	64K bytes, 4-way associative, 32-byte blocks, four read/write ports
Unified L2 Cache	1M bytes, 4-way associative, 64-byte blocks
Branch predictor	Hybrid local/global (like 21264): 13-bit global history register indexing 8K-entry global PHT and 8K-entry choice table; 2K 11-bit local history regs indexing 2K local PHT; 4K-entry BTB, 16-entry RAS (per thread)
Fetch/Decode/Issue/Commit Width	8 instructions/cycle
Function Units	6 Int ALU, 2 Int Multiply, 4 FP Add, 2 FP Multiply
Fetch to Decode Latency	5 cycles
Decode to Execution Latency	10 cycles

The LVQ provides a couple of additional advantages. The LVQ reduces the pressure on the data cache ports because, unlike the ALAB design, only one thread probes the data cache. Additionally, the LVQ can accelerate fault detection of faulty addresses by comparing the effective address of the leading thread from the LVQ with the effective address of the trailing thread. However, the LVQ constrains the scheduling of trailing-thread loads, as each load must occur in program order with respect to other trailing-thread loads and after the corresponding leading-thread load to retrieve correct values from the queue.

3.4 Two Techniques to Enhance Performance

An SRT processor can improve its performance using one thread to improve cache and branch prediction behavior for the other thread. In this paper, we evaluate two such mechanisms—*slack fetch* and *branch outcome queue*—that allow the trailing thread to benefit from the first thread's execution.

The slack fetch mechanism tries to maintain a constant slack of instructions between the leading and trailing thread. Ideally, branches from the trailing thread should probe the branch predictor after the corresponding branch from the leading thread has executed and updated the predictor. Similarly, accesses from the trailing thread should probe the instruction and data caches after the corresponding accesses from the leading thread have incurred any misses and brought in the cache blocks. The slack fetch mechanism relies on a signed counter that is decremented when the (pre-designated) leading thread commits an instruction and incremented when the (pre-designated) trailing thread commits an instruction. The counter is initialized at reset to the target slack. By adding the counter's value in each cycle to the trailing thread's instruction count, the ICount fetch policy (see Section 2.2) favors the leading thread until the target slack is reached, automatically guiding the fetch stage to maintain the desired slack.

The branch outcome queue reduces misspeculation more directly and effectively than the slack fetch mechanism. This technique uses a hardware queue to deliver the leading thread's committed branch outcomes (branch PCs and target addresses) to the trailing thread. In the fetch stage, the trailing thread uses the head of the queue much like a branch target buffer, reliably directing the thread down the same path as the leading thread. Consequently, in the absence of faults, the trailing thread's branches never misfetch or mispredict and the trailing thread never misspeculates. To keep the threads in sync, the leading thread stalls in the commit stage if it cannot retire a branch because the queue is full, and the trailing thread stalls in the fetch stage if the queue becomes empty.

Table 2. Benchmark Statistics for Baseline Architecture. (inst = instructions, ld = load, st = store, B = billion, M = million, Occ = occupancy, Acc = accuracy)

Bench-mark	Input	inst skip-ped (B)	ld (M)	st (M)	Base IPC	RUU Occ	Br Pred Acc (%)
compress	ref	10	56.7	32.7	2.33	133	93.6
fpppp	ref	20	76.9	26.6	4.46	167	98.0
gcc	amptjp.i	0.3	64.6	31.7	1.64	73	92.2
go	9stone21	10	58.7	21.0	1.19	78	77.2
hydro2d	ref	20	39.9	8.9	2.53	229	99.8
ijpeg	penguin	10	38.2	17.7	3.02	140	91.0
li	ref	10	69.7	35.6	2.55	107	96.8
m88ksim	ref	10	47.1	30.7	3.33	120	97.1
perl	primes	10	68.7	43.5	2.19	65	100
tomcatv	ref	20	41.0	10.4	3.46	221	98.9
vortex	ref	20	65.8	36.6	4.05	174	99.6

Our results show that slack fetch and branch outcome queue improve performance by 10% and 14% on our benchmarks, respectively. The combination of the two improves performance by as much as 27% (15% on average).

4. EVALUATION METHODOLOGY

In this section, we describe our simulator, baseline SMT architecture, and benchmarks. Our simulator is a modified version of the "sim-outorder" simulator from the Simplescalar tool set [2]. We extended the original simulator by replicating the necessary machine context for SMT, adding support for multiple address spaces, and increasing the coverage and types of collected statistics.

Our baseline SMT architecture is an aggressive, out-of-order, and speculative processor. Table 1 summarizes key parameters of the processor. We simulate a pipeline with a long front-end pipeline to account for overhead due to SMT, complexity of an out-of-order machine, and wire delays in future high-frequency microprocessors. To approximate the performance of a processor employing a trace cache [9], we fetch up to three basic blocks (up to a maximum of eight instructions) per cycle regardless of location.

We evaluated our ideas using 11 SPEC95 benchmarks [17] shown in Table 2. We compiled all benchmarks for the SimpleScalar PISA instruction set using gcc 2.6 with full optimization. We ran all benchmarks for 200 million instructions using inputs from the SPEC reference data set. Table 2 shows the number of initial instructions skipped (using SimpleScalar 3's checkpointing mechanism), the number of executed loads and stores, single-thread IPC, the average number of entries in the instruction window (RUU), and the branch predictor accuracy for each benchmark.

5. RESULTS

This section studies an SRT processor's performance in the absence of transient hardware faults. This analysis helps us understand the performance impact of constraints introduced by output comparison and input replication. However, we do not examine an SRT processor's performance in the presence of faults because that depends on the specific fault recovery scheme, which is beyond the scope of this paper. Also, because faults are relatively infrequent, the performance of an SRT processor is determined predominantly by times when there are no faults.

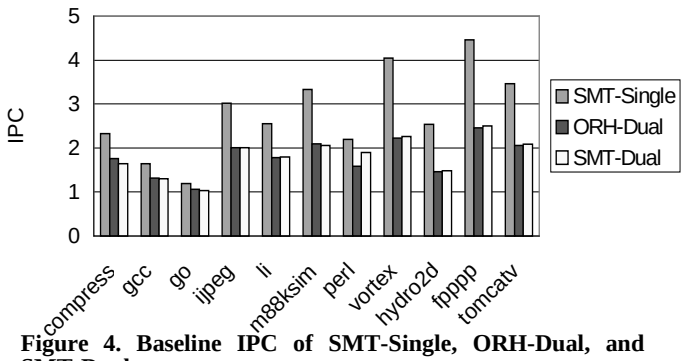

Figure 4. Baseline IPC of SMT-Single, ORH-Dual, and SMT-Dual.

We first present results of our baseline architectures against which we will compare our SRT results (Section 5.1). Then, we show that, in the absence of any constraints of output comparison and input replication, our two performance enhancement techniques—slack fetch and branch outcome queue—significantly boost an SRT processor's performance (Section 5.2). Section 5.3 and Section 5.4 will add the constraints of output comparison and input replication, respectively, to the results in Section 5.2 and show that neither our output comparison techniques nor our input replication techniques adversely affect performance. Finally, Section 5.5 summarizes our overall results.

5.1 Baseline Characterization

This section compares the baseline IPCs of our three base machines: SMT-Single, ORH-Dual, and SMT-Dual. SMT-Single is our base SMT machine running one thread, whereas SMT-Dual is the same machine running two copies of the same program simultaneously. SMT-Dual's replicated threads run in the same address space; however, SMT-Dual does not have any fault detection hardware. SMT-Dual helps us understand the performance of two copies of the same thread running on our SMT machine without any output comparison, input replication, or performance enhancement techniques. In the next three sections, we will develop the SMT-Dual machine into an SRT processor.

ORH-Dual is a processor with two pipelines, each of which runs a copy of the same program. One or more checkers check the pipeline outputs for faults. We assume that ORH-Dual's checker executes in zero cycles and incurs no performance penalty, even though in practice a checker could take multiple cycles to compare outputs from the replicated threads in ORH-Dual. We assume that instructions are replicated in the fetch stage, so (like SMT-Single) up to eight instructions can be fetched per cycle and forwarded to both pipelines. We configured each of ORH-Dual's execution pipelines to have half the resources of the SMT pipeline: half the RUU and LSQ slots and half the function units and data cache ports. This allows us to compare the performance of an on-chip hardware-replicated solution (such as the IBM G5 microprocessor) against an SRT processor. Because ORH-Dual does not replicate the fetch stage, its branch predictor is identical to the one in our SMT machine.

Figure 4 shows that on average ORH-Dual and SMT-Dual have roughly the same performance. However, both are 32% slower on average than SMT-Single. Note that for perl, SMT-Dual is slightly faster than ORH-Dual. In this case, SMT-Dual's branch target buffer (BTB) delivers around 2% more good branch target address predictions than ORH-Dual's BTB, possibly because one thread prefetches BTB entries for the other. In Section 5.2, we explore

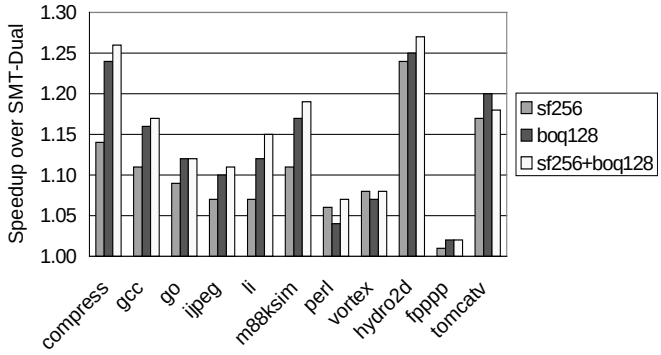

Figure 5. Impact of Slack Fetch and Branch Outcome Queue on SMT-Dual. sf256 = Slack Fetch with a slack of 256 instructions, boq128 = Branch Outcome Queue with 128 entries, sf256+boq128 = combination of the two.

the slack fetch and branch outcome queue optimizations that seek to exploit such effects further.

5.2 Slack Fetch and Branch Outcome Queue

In this section we examine the performance of slack fetch and branch outcome queue for an SMT-Dual machine. Slack fetch tries to maintain a constant difference between the number of instructions fetched from the leading and trailing thread, hoping to maximize implicit prefetching effects in the caches and branch prediction structures. The branch outcome queue explicitly feeds the leading thread's branch outcomes to the trailing thread; in the absence of faults, the trailing thread never misspeculates.

Figure 5 shows that with a slack of 256 instructions between the leading and trailing thread, slack fetch improves the IPC of the SMT-Dual machine by 10% (averaged across 11 benchmarks). A 128-entry branch outcome queue improves performance by 14% on average. Finally, slack fetch and the branch outcome queue together provide a performance boost of 15%.

Both techniques improve performance for the same two reasons. First, both significantly reduce the amount of memory stalls seen by the trailing thread, because the leading thread prefetches cache misses for the trailing thread. The branch outcome queue prevents the trailing thread from fetching past a branch instruction until after the leading thread commits the corresponding dynamic branch, forcing a time delay between the threads even without the slack fetch mechanism. On average across all benchmarks, the leading thread was 107 instructions ahead of the trailing thread for boq128, compared with 179 for sl256 and 162 for sf256+boq128. As a result, sf256, boq128, and sf256+boq128 reduce the dispatch to retire latency of loads from the trailing thread by 17%, 25%, and 25% respectively (averaged across all loads in the trailing thread). Thus the second thread sees significantly reduced memory stall times, even though it does not necessarily see fewer cache misses.

Second, both techniques reduce the amount of misspeculation due to branch mispredictions and branch target buffer misses. With slack fetch, branches from the trailing thread encounter fewer branch mispredictions because corresponding branches from the leading thread have enough time to execute and update the branch predictor and branch target buffer. Thus, sf256 reduces the number of instructions squashed due to mis-speculation by 50% on average. In contrast, the branch outcome queue eliminates all misspeculation from the trailing thread by design, which typically leads to better performance. Note that in the cases of perl and

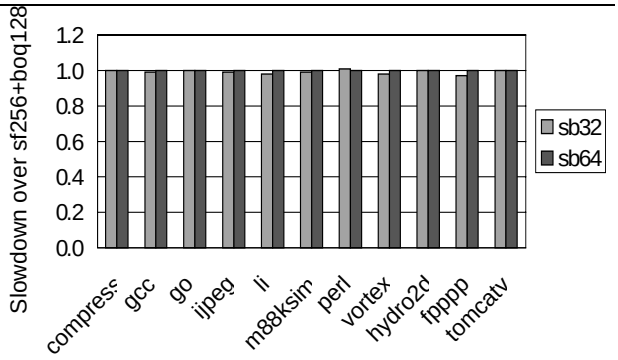

Figure 6. Slowdown due to Output Comparison of Stores. This figure shows the slowdown due to output comparison of stores in our first sphere of replication (Section 3.2) over sf256+boq128 (Figure 5). Sb32 = sf256+boq128 with 32 store buffer entries. Sb64 = sf256+boq128 with 64 store buffer entries.

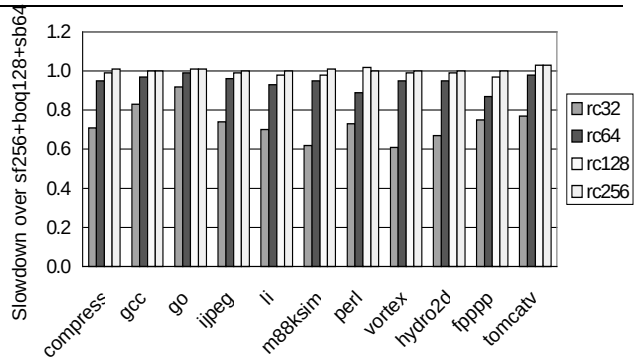

Figure 7. Slowdown due to Output Comparison of Register Writebacks. The vertical axis shows the relative IPC of adding a register check buffer with N entries (rcN) to a base configuration of sf256+boq128+sb64 (Figure 6).

vortex, sf256 performs better than boq128 because sf256 allows branches to execute out of order and speculatively in the trailing thread, making more effective use of available processor resources.

Finally, we found that slack length and branch outcome queue size have a secondary effect on performance. Slack fetch with slack of 256 instructions improves performance over slack of 32 and 128 instructions by 8% and 3% respectively. Increasing the target slack further does not affect performance significantly. For slack values up to 4096 instructions, performance changes less than 5% for each benchmark (improving for some, degrading for others) relative to a 256-instruction slack. The branch outcome queue could be made as small as 32 entries without affecting performance. Nevertheless, we chose a branch outcome queue of 128 entries for our subsequent experiments in Section 5.3 and Section 5.4 to avoid stalls due to full conditions in the branch outcome queue. Also, the branch outcome queue with 128 entries would occupy roughly 1K bytes, with each entry having 4 bytes for the PC and 4 bytes for the branch address.[4] Clearly, a 128-entry branch outcome queue is quite small and will not stress a microprocessor's transistor budget.

5.3 Output Comparison

In this section, we study the performance degradation due to output comparison of stores and register writebacks in our two spheres of replication (Section 3.2). In this section we assume that inputs are replicated by an oracle. In the next section, we will study the performance impact of input replication. Figure 6 shows the slowdown due to output comparison of stores from our first sphere of replication. We assume a base machine with slack fetch (slack length = 256 fetched instructions) and branch outcome queue (queue size = 128 entries). We perform output comparison of stores in a non-coalescing store buffer, which is shared between the two replicated threads. Stores from the leading thread are appended to the buffer as they retire from the RUU. Trailing-thread stores enter the same store buffer on retirement and are compared with the corresponding store that must already be present. A single store is then issued to the cache and retired from the buffer when completed. Thus, the leading thread may stall if it runs out of store buffer entries, while the trailing thread tries to catch up and retire corresponding stores from the store buffer. However, Figure 6 shows that a 64-entry store buffer eliminates almost all such stalls. Even a 32-entry store buffer is sufficient in most cases; across all our benchmarks, it slows down the machine by only 1% on average.

Figure 7 shows the performance degradation for our second sphere of replication. The base machine assumes slack fetch (slack length = 256 fetched instructions), branch outcome queue (with 128 entries) and a 64-entry store buffer. Output comparison of register writebacks can stall the leading thread and degrade performance, if the register check buffer fills up frequently and the trailing thread tries to catch up with the leading thread. The figure shows that performance degradation is eliminated with a register check buffer size of 256 entries. In contrast, 32-entry, 64-entry, and 128-entry register check buffers degrade performance by 27%, 6%, and 1% respectively (on average). This result follows from the fact that the slack fetch mechanism is trying to maintain a 256-instruction separation between the threads.

Thus, in our simulated machine, a 64-entry store buffer and a 256-entry register check buffer eliminate all performance degradation due to output comparison in both spheres of replication.

5.4 Input Replication

In this section we study the impact of performance degradation due to the active load address buffer (ALAB) and load value queue (LVQ) (Section 5.4). In this section, we assume that output replication is performed by an oracle. In Section 5.5, we will show our overall results with both output comparison and input replication.

Figure 8 shows that both an ALAB with 64 entries and an LVQ with 64 entries pay almost no performance penalty due to input replication. The load scheduling restrictions of the LVQ do not appear to have significant impact. However, typically, benchmarks with a higher fraction of loads (e.g., li) have a higher performance degradation with a 16-entry ALAB or a 16-entry LVQ. On average a 16-entry ALAB and a 16-entry LVQ degrade performance by 8% and 5% respectively.

5.5 Overall Results

Finally, Figure 9 shows the overall improvement of an SRT processor with slack fetch (with slack of 256 instructions), branch outcome queue (with 128 entries), store output comparison (with a 64-entry store buffer), and load value queue (with 64 entries). This figure shows that such an SRT processor improves performance

[4] Because the BOQ is merely a performance optimization, fault protection is not required, although it would be simple to add.

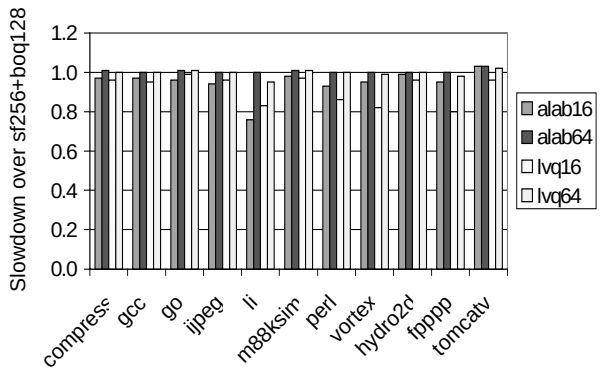

Figure 8. Slowdown due to Input Replication. alabN = sf256+boq128 with an active load buffer with N entries, lvqN = sf256+boq128 with a load value queue with N entries.

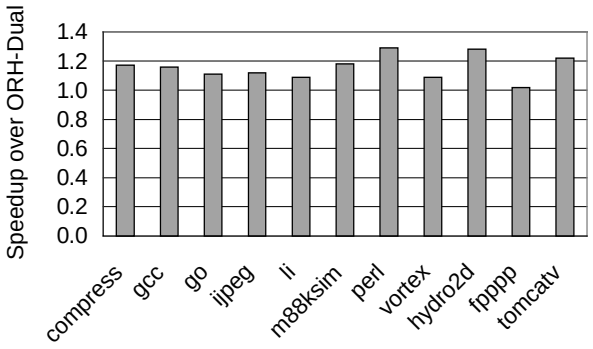

Figure 9. Speedup of sf256+boq128+sb64+lvq64 over ORH-Dual.

over ORH-Dual—an on-chip hardware-replicated solution that detects hardware faults via cycle-by-cycle lockstepping—by 16% on average, with a maximum improvement of up to 29%. The case in which the SRT processor has the smallest advantage, fpppp, is a program that has a low branch misprediction and cache miss rates, and in fact is primarily limited by the number of floating-point function units even in the SMT-Single case. In general, an SRT processor provides significantly better performance than an equivalently sized on-chip, hardware-replicated solution.

6. IMPROVING COVERAGE OF MULTI-CYCLE FAULTS

Thus far, we have assumed that transient faults are of very short duration, so that simple time redundancy is as effective as space redundancy for fault detection. However, some faults may persist for multiple cycles, such that two redundant operations could pass through the same hardware during the duration of the fault and experience the same error. In this case, output comparison would not detect the fault, and it would be silently propagated outside the sphere of replication, potentially causing a loss of data integrity. We briefly note three techniques for improving the ability of an SRT processor to detect multi-cycle transient and some permanent faults.

First, the slack fetch mechanism can be used to control the time between corresponding instructions from the redundant threads. Increasing the slack between threads increases the time between corresponding instructions, reducing the probability that these instructions will encounter the same transient fault. The quantitative effect of varying the target slack on fault coverage is an area of future work.

Second, we can increase the spatial redundancy of an SRT processor by statically partitioning function units into two groups and guaranteeing that corresponding instructions from redundant threads are executed in opposite groups. For example, we can take the least-significant bit of each instruction's program counter value (e.g., bit 2 for a machine with 32-bit instructions) and exclusive-or it with the thread ID (0 or 1 for the two redundant threads). Based on this bit value, we can assign the instruction to one of two clusters of execution units. Corresponding instructions will thus be assigned to opposite clusters, providing detection of any transient or permanent fault that is confined to one cluster. Applying this technique to our (otherwise unmodified) final SRT configuration from the previous section caused a performance loss of less than 2% across all but one of the benchmarks. The exception, hydro2d, apparently has an imbalanced distribution of instruction types among "even" and "odd" PCs, and its performance dropped by 15%. We were able to reduce its performance loss to under 2% by using the parity of the five least-significant PC bits instead of a single PC bit to determine each instruction's placement.

Finally, we can apply the technique of shifted operands [7] to force corresponding operations through different bit lanes of busses and function units. For example, busses and logic could be 65 bits wide, with operands of one thread using bits 0-64 and operands of the other bits 1-65. Even if corresponding redundant instructions experienced the same fault, it would very likely affect different bits of the result, leading to a detectable fault.

7. RELATED WORK

Rotenberg's AR-SMT design [11] was the first to use simultaneous multithreading to detect transient hardware faults. AR-SMT incorporates two redundant threads: the "active", or A-thread, and the "redundant", or R-thread. Committed register writebacks and load values from the A-thread are placed in a *delay buffer*, where they serve as the alternate execution stream against which R-thread results are checked and predictions to eliminate speculation on the R-thread. Thus the delay buffer combines our register check buffer and branch outcome queue. In addition, unlike our designs, the R-thread uses the delay buffer as a source of value predictions to speculate past data dependencies.

AR-SMT is one point in the SRT design space; its sphere of replication is similar to our second sphere in which the register file resides outside. In AR-SMT, the R-thread register file serves as the architectural file: register writeback values are verified before updating the R-thread registers, and the R-thread file is considered to be a valid checkpoint for fault recovery. As with the register files in our second sphere, the A-thread register file serves only to bypass uncommitted register updates still in the delay buffer. Thus replication does not provide fault coverage for the R-thread register file, so this register file must be augmented with an alternate coverage technique, such as ECC. Otherwise, a fault in an R-thread register value would lead to a mismatch in A-thread and R-thread results. AR-SMT would correctly detect this fault, but may improperly recover by restarting from the corrupted R-thread register file contents.

Our first sphere of replication shows that, unlike AR-SMT, fault detection can be achieved in SRT processors without checking the result of every instruction, although at the loss of a protected architectural register file that can be used for fast recovery.

AR-SMT varies significantly from our SRT designs in that all of main memory is inside the sphere of replication. This scheme

provides better memory fault detection than ECC. Nevertheless, doubling the physical memory of a system can be very expensive. This technique also halves effective cache capacities. As a result, most commercial systems protect the memory system with ECC. We demonstrate that SRT processors can use the same solution.

AR-SMT does not deal with input replication or output comparison for values that enter or exit the sphere of replication via uncached accesses or via main memory through DMA I/O or multiprocessor accesses. Failure to properly replicate inputs could cause divergent execution leading to false faults, while failure to compare I/O outputs could allow corrupted data to leave the system.

AR-SMT also requires operating system modifications to manage the additional address mappings needed to replicate the address space. The design disables redundant threading on OS calls, leaving kernel code vulnerable to transient hardware faults. In contrast, our SRT designs perform replication in hardware, providing transparent and continuous fault coverage.

Finally, our performance evaluation complements Rotenberg's, as he used a trace processor model [10] while we based ours on a more conventional superscalar design.

Recently, Austin proposed a novel fault detection architecture called DIVA [1]. DIVA uses a very simple in-order processor as a checker for a larger out-of-order, speculative processor. The DIVA architecture has both advantages and disadvantages compared to SRT processors. Unlike an SRT thread, DIVA's checker is a completely separate processor. Consequently, DIVA's checker can detect permanent faults and design errors in the main processor as well as transient faults. Additionally, unlike SRT processors, the DIVA architecture does not require an SMT architecture for fault detection.

On the other hand, SRT processors have three advantages over the DIVA architecture. First, unlike DIVA, which requires designing the checker from scratch, an SRT processor requires modest hardware modification over an SMT processor, such as the Alpha 21464 [3]. Thus, an SRT processor can leverage existing SMT designs. A single device may even be configurable as an SRT or SMT processor, trading fault tolerance for throughput depending on the target market.

Second, in the presence of DMA I/O devices or multiple processors, the DIVA architecture, like AR-SMT, can detect false transient faults due to external memory writes (see Section 3.3.1). We believe that the DIVA architecture could be augmented with either an ALAB or LVQ to avoid this problem.

Third, to guarantee forward progress in the presence of permanent faults in the main processor, DIVA assumes that the checker is always correct, and proceeds using the checker's result in the case of a mismatch. As a result, faults in the checker itself—including transient faults—must be detected or avoided through alternative techniques. (For example, Austin suggests that design errors could be avoided by formally verifying the checker.) Although SRT processors cannot detect design faults, transient faults in either thread will be detected. Fault detection in checking logic itself still requires additional techniques, but we believe the amount of SRT checking logic would be much more limited than the DIVA checker.

All three fault detection architectures—AR-SMT, DIVA, and SRT—are specific implementations of a more general concept called *watchdog processors* [6]. A watchdog processor is one that runs concurrently with the main processor, observes the main processor's outputs and inputs, and compares its own outputs (either pre-computed or concurrently computed) with the main processor's outputs. For DIVA, the checker is the watchdog processor. For AR-SMT and SRT, one of the redundant threads serves as the watchdog processor.

As we discussed earlier, the Compaq NonStop Himalaya [20] and IBM S/390 G5 microprocessor [13] are commercial fault-tolerant computers that employ cycle-by-cycle lockstepping to detect hardware faults. In this paper, we demonstrate that an SRT processor can provide similar transient fault coverage, but with superior performance.

Finally, there is a rich body of research (e.g., [6][4][8][7][15][5]) that examines fault detection using time or space redundancy of functional units. We improve upon this work in two ways. First, our definition of the sphere of replication allows an SRT processor to perform output comparison of selected instructions in the committed instruction stream. Further, an SRT processor does not perform output comparison of speculative instructions that get squashed. In contrast, time- or space-redundant functional units must typically check every instruction that passes through them for faults. Second, we can perform performance optimizations, such as slack fetch and branch outcome queue, because an SRT processor employs thread-level fault detection, as opposed to operation-level fault detection.

8. CONCLUSION

Future processors will be increasingly prone to transient hardware faults due to smaller feature sizes, reduced voltage levels, higher transistor counts, and reduced noise margins. Most commercial fault-tolerant computers use hardware replication and lockstepping to detect such hardware faults in microprocessors. Unfortunately, this static partitioning of hardware resources among replicated components is a source of inefficiency.

In this paper, we demonstrated that a Simultaneous and Redundantly Threaded (SRT) processor—derived from a Simultaneous Multithreaded (SMT) processor—provides similar levels of transient fault coverage with significantly higher performance than static physical replication. An SRT processor provides transient fault coverage by running identical copies of the same program simultaneously. An SRT processor can provide higher performance because it can dynamically schedule its hardware resources among the redundant copies. Unfortunately, dynamic scheduling of hardware resources also makes it impossible to use lockstepping in an SRT processor because the same instructions from redundant threads may not execute in the same cycle or in the same order.

This paper makes four contributions to the design of SRT processors. First, we introduced the sphere of replication, which abstracts both the physical redundancy of a lockstepped system and the logical redundancy of an SRT processor. This framework aids in identifying the scope of fault coverage and the output and input values requiring special handling. Second, we identified two viable spheres of replication in an SRT processor, one of which provides complete fault detection while checking only committed stores and uncached loads (but not every instruction result). Third, we identified the need for consistent replication of load values, and proposed and evaluated two new mechanisms—the active load address buffer and the load value queue—for satisfying this requirement. Finally, we proposed and evaluated two mechanisms—slack fetch and branch outcome queue—that enhance the performance of an SRT processor by allowing one thread to benefit from the prior execution of the other thread. Our results with 11 SPEC95 benchmarks show that an SRT processor can outperform an equivalently sized, on-chip, hardware-

replicated solution by 16% on average, with a maximum benefit of up to 29%.

ACKNOWLEDGMENTS

We thank Bob Jardine and Alan Wood from Compaq's Tandem Division for our numerous discussions with them on fault tolerance and their encouragement to pursue this research. We thank Steven E. Raasch for developing the SMT version of Simplescalar on which this work is based. We thank Bill Bowhill, Joel Emer, Roger Espasa, Tryggve Fossum, Mark Hill, Toni Juan, Rick Kessler, Geoff Lowney, Todd Mowry, Andre Seznec, Jim Smullen, and our anonymous referees for their insightful comments on different drafts of this paper.

REFERENCES

[1] Todd M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," Proceedings of the 32nd Annual International Symposium on Microarchitecture, November 1999.

[2] D. A. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report #1342, University of Wisconsin-Madison, Computer Sciences Department, June 1997.

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

[4] Manoj Franklin, "Incorporating Fault Tolerance in Superscalar Processors," Proceedings of High Performance Computing, December, 1996.

[5] John G. Holm and Prithviraj Banerjee, "Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions," Proceedings of the International Conference on Parallel Processing, 1992.

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

[7] 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, July 1982.

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

[9] Eric Rotenberg, Steve Bennett, and James E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," Proceedings of the 29th Annual International Symposium on Microarchitecture, pp 24–34, December 1996.

[10] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith, "Trace Processors," 30th Annual International Symposium on Microarchitecture (MICRO-30), Dec 1997.

[11] Eric Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," Proceedings of Fault-Tolerant Computing Systems (FTCS), 1999.

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

[13] T.J.Slegel, et al., "IBM's S/390 G5 Microprocessor Design," IEEE Micro, pp 12–23, March/April, 1999.

[14] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans. on Computers, 37(5):562–573, May 1988.

[15] 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 International Symposium on Fault-Tolerant Computing, pp. 436–443, 1989.

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

[17] SPEC newsletter. Fairfax, Virginia, September 1995.

[18] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proceedings of the 22nd Annual International Symposium on Computer Architecture, Italy, June 1995.

[19] 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," Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA), May, 1996.

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

[21] Wayne Yamamoto and Mario Nemirovsky, "Increasing Superscalar Performance Through Multistreaming," Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT), pp 49–58, June 1995.