Transient-Fault Recovery Using Simultaneous Multithreading

T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng
School of Electrical and Computer Engineering, Purdue University, W. Lafayette, IN 47907
{vijay, pomeranz, kkcheng}@ecn.purdue.edu

Abstract

*We propose a scheme for transient-fault recovery called **Simultaneously and Redundantly Threaded processors with Recovery (SRTR)** that enhances a previously proposed scheme for transient-fault detection, called Simultaneously and Redundantly Threaded (SRT) processors. SRT replicates an application into two communicating threads, one executing ahead of the other. The trailing thread repeats the computation performed by the leading thread, and the values produced by the two threads are compared. In SRT, a leading instruction may commit **before** the check for faults occurs, relying on the trailing thread to trigger detection. In contrast, SRTR must **not** allow **any** leading instruction to commit before checking occurs, since a faulty instruction cannot be undone once the instruction commits.*

*To avoid stalling leading instructions at commit while waiting for their trailing counterparts, SRTR exploits the time between the completion and commit of leading instructions. SRTR compares the leading and trailing values as soon as the trailing instruction completes, typically before the leading instruction reaches the commit point. To avoid increasing the bandwidth demand on the register file for checking register values, SRTR uses the **register value queue (RVQ)** to hold register values for checking. To reduce the bandwidth pressure on the RVQ itself, SRTR employs **dependence-based checking elision (DBCE)**. By reasoning that faults propagate through dependent instructions, DBCE exploits register (true) dependence chains so that only the last instruction in a chain uses the RVQ, and has the leading and trailing values checked. SRTR performs within 1% and 7% of SRT for SPEC95 integer and floating-point programs, respectively. While SRTR without DBCE incurs about 18% performance loss when the number of RVQ ports is reduced from four (which is performance-equivalent to an unlimited number) to two ports, with DBCE, a two-ported RVQ performs within 2% of a four-ported RVQ.*

1 Introduction

The downscaling of feature sizes in CMOS technologies is resulting in faster transistors and lower supply voltages. While this trend contributes to improving the overall performance and reducing per-transistor power, it also implies that microprocessors are increasingly more susceptible to transient faults of various types. For instance, cosmic alpha particles may charge or discharge internal nodes of logic or SRAM cells; and lower supply voltages result in reduced noise margins allowing high-frequency crosstalk to flip bit values. The result is degraded reliability even in commodity microprocessors for which reliability has not been a concern until recently.

To address reliability issues, Simultaneously and Redundantly Threaded (SRT) processors are proposed in [10] based on the Simultaneous Multithreaded architecture (SMT) [17]. SRT detects transient faults by replicating an application into two communicating threads, one (called the leading thread) executing ahead of the other (called the trailing thread). The trailing thread repeats the computation performed by the leading thread, and the values produced by the two threads are compared.

Although SRT does not support recovery from faults, the following features introduced by SRT for fault detection [10] are important for recovery as well: (1) Replicating cached loads is problematic because memory locations may be modified by an external agent (e.g., another processor during multiprocessor synchronization) between the time the leading thread loads a value and the time the trailing thread tries to load the same value. The two threads may diverge if the loads return different data. SRT allows only the leading thread to access the cache, and uses the Load Value Queue (LVQ) to hold the leading load values. The trailing thread loads from the LVQ instead of repeating the load from the cache. (2) The leading thread runs ahead of the trailing thread by a long *slack* (e.g., 256 instructions), and provides the leading branch outcomes to the trailing thread through the Branch Outcome Queue (BOQ). The slack and the use of branch outcomes hide the leading thread's memory latencies and branch mispredictions from the trailing thread, since by the time the trailing thread needs a load value or a branch outcome, the leading thread has already produced it. (3) By replicating register values but not memory values, SRT compares *only* stores and uncached loads, but not register values, of the two threads. Because an incorrect value caused by a fault propagates through computations and is eventually consumed by a store, checking *only* stores suffices.

We propose *Simultaneously and Redundantly Threaded processors with Recovery (SRTR)* to extend SRT to include recovery. Although systems using software recovery often employ hardware detection [3,13], software checkpointing incurs considerable performance cost even when there are no faults. Therefore, hardware recovery at a modest performance cost over detection is an attractive option, especially because hardware recovery permits the use of off-the-shelf software. We identify, for the first time, the following key issues:

- **Problem:** A fundamental implication of the SRT detection scheme is that a leading non-store instruction may commit *before* the check for faults occurs, relying on the trailing thread to trigger detection. SRTR, on the other hand, must *not*

1063-6897/02 $17.00 © 2002 IEEE

allow *any* leading instruction to commit before checking occurs, since a faulty instruction cannot be undone once the instruction commits. Unless care is taken, leading instructions will stall at commit waiting for their trailing counterparts to complete and undergo checking. This stalling will create pressure on the instruction window and physical registers, and degrade performance.

- **Solution:** To avoid stalling leading instructions, SRTR exploits the time between the completion and commit of leading instructions. SRTR checks the results of a leading and the corresponding trailing instruction as soon as the trailing instruction completes, well before the leading instruction reaches the commit point. For the SPEC95 benchmarks, complete to commit times average at 29 cycles. This gap provides sufficient time for a trailing instruction to complete before the leading instruction reaches the commit point. To exploit the complete to commit time, the slack between the leading and trailing threads in SRTR must be short. At the same time, a slack that is too short would cause the trailing thread to stall due to unavailable branch outcomes and load values from the leading thread. To support an appropriately short slack, SRTR's leading thread provides the trailing thread with branch predictions instead of outcomes. Because the leading thread's branch predictions are available much earlier than the branch outcomes, and because a short slack is sufficient for hiding on-chip cache hit latencies, SRTR avoids trailing thread stalls *even* with a short slack. We show that high prediction accuracies and low off-chip miss rates in the underlying SMT enable SRTR, using a slack of 32, to perform within 5% of SRT, using a slack of 256 (as in [10]), when the recovery mechanisms of SRTR are disabled so that both schemes target only detection.
- **Problem:** By the time a leading instruction reaches the commit point, its register value often has been written back to the physical register file. Because *all* instructions are checked in recovery, accessing the register file in order to perform the check will add substantial bandwidth pressure.
- **Solution:** SRTR uses a separate structure, the *register value queue (RVQ),* to store register values and other information necessary for checking of instructions, avoiding bandwidth pressure on the register file.
- **Problem:** There is bandwidth pressure on the RVQ.
- **Solution:** We propose *dependence-based checking elision (DBCE)* to reduce the number of checks, and thereby, the RVQ bandwidth demand. By reasoning that faults propagate through dependent instructions, DBCE exploits register (true) dependence chains so that *only* the last instruction in a chain uses the RVQ and has leading and trailing values checked. The chain's earlier instructions in *both* threads completely elide the RVQ. SRT can be viewed as taking such elision to the extreme by observing that stores are the last instructions in any register dependence chain, and that only stores need to be checked. However, SRT's chains are too long for SRTR because the leading thread cannot commit until the last instruction in the long chain is checked. DBCE forms short chains by exploiting the abundant register dependencies in near-by instructions. Because of the short slack and short chains, the trailing chain's last instruction completes only a few cycles behind the leading chain's earlier instructions. Consequently, checking of the last instruction is usually done between the time the earliest leading instruction completes and the time it reaches the commit point. DBCE redundantly builds chains in both threads and checks its own functionality for faults. DBCE elides 35% of checks for the benchmarks we consider. SRTR performs within 1% and 7% of SRT for SPEC95 integer and floating-point programs, respectively. SRTR without DBCE incurs about 18% performance loss when the number of RVQ ports is reduced from four (which is performance-equivalent to an unlimited number) to two ports. With DBCE, a 2-ported RVQ performs within 2% of a 4-ported RVQ.

SRTR is guaranteed to provide recovery from single transient faults except those that affect the register file and that fail to propagate through dependence chains, in which case SRTR guarantees detection.

We review SRT in Section 2. We describe the SRTR scheme in Section 3 and DBCE in Section 4. In Section 5, we present experimental results. In Section 6, we discuss related work, and conclude in Section 7.

2 Transient-fault detection: background

SRT uses SMT's multithreaded execution to replicate an application into two communicating threads, one executing ahead of the other. Comparing the results of two redundant executions is the underlying scheme to detect transient faults in SRT. Because detection is based on replication, the extent to which the application is replicated is important. SRT formalizes this notion by defining the *sphere of replication (SoR)* [10], pointing out that (1) all computation and data within this sphere are replicated such that each thread uses its own copy, (2) data entering the SoR is independently read by the two threads using *input replication*, (3) data exiting the SoR from the two threads are compared using *output comparison*, and only one copy of the checked data is stored outside the SoR. Note that the input replicator and output comparator are outside the SoR. The input replicator and output comparator must be self-checked and are typically implemented using dual-rail logic. Because data outside the SoR is not replicated, other forms of protection such as ECC are needed outside the SoR.

Two possible SoRs defined in [10] are: processor and registers inside the SoR, with the cache hierarchy outside; and only the processor inside the SoR, with the registers and cache hierarchy outside. In the first SoR, leading and trailing thread values need to be compared only for stores and uncached loads, while other instructions can commit without comparing values. SRT uses a store buffer (StB) in which the leading thread places its committed store values and addresses. The trailing thread's store values and addresses are compared against the StB entries to determine whether a fault has occurred. In the second SoR, all register values need to be checked. Checking registers requires large buffers with high bandwidth to avoid performance degradation.

As explained in Section 1, to handle cached loads, SRT uses the ECC-protected Load Value Queue (LVQ) in which the

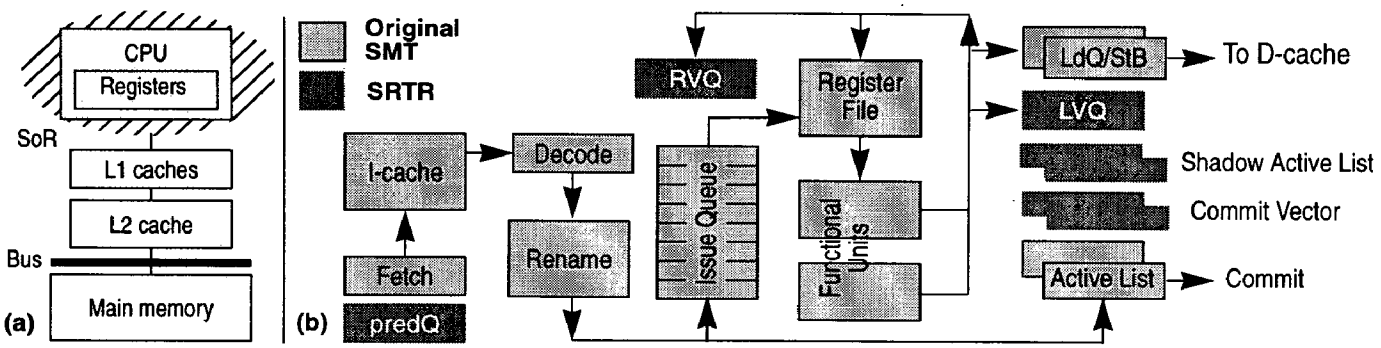


FIGURE 1: (a) SRTR's Sphere of Replication. (b) SRTR's additions to SMT.

leading thread deposits its committed load values and addresses. The trailing thread obtains the load value from the LVQ, instead of repeating the load from the memory hierarchy. The Active Load Address Buffer proposed in [10] is an alternative for the LVQ that also addresses this problem. We use the LVQ because it is simpler.

A key optimization in SRT is that the leading thread runs ahead of the trailing thread by an amount called the *slack* (e.g., the slack may be 256 instructions). In addition, the leading thread provides its branch outcomes via the branch outcome queue (BOQ) to the trailing thread. The slack and the communication of branch outcomes hide the leading thread's memory latencies and avoids branch mispredictions from the trailing thread. Due to the slack, by the time the trailing thread needs a load value or branch outcome, the leading thread has already produced it.

SRT assumes that uncached accesses are performed non-speculatively. SRT synchronizes uncached accesses from the leading and trailing threads, compares the addresses, and replicates the load data. SRT assumes that code does not modify itself, although self-modifying code in regular SMTs already requires thread synchronization and cache coherence which can be extended to keep the leading and trailing threads consistent. For input replication of external interrupts, SRT suggests forcing the threads to the same execution point and then delivering the interrupt synchronously to both threads.

3 Transient-fault recovery

We propose *Simultaneously and Redundantly Threaded processors with Recovery (SRTR)* that enhances SRT to include transient-fault recovery. A natural way to extend SRT to achieve recovery is to use the rollback ability of modern pipelines, which is provided to support precise interrupts and speculative execution [9]. Because transient faults do not persist, rolling back to the offending instruction and re-executing guarantees forward progress.

In SRT, a leading non-store instruction may commit *before* the check for faults occurs. SRTR, on the other hand, must *not* allow *any* instruction to commit before it is checked for faults. Trailing instructions must complete and results must be compared as much before the leading instructions reach the commit point as possible, so that leading instructions do not stall at commit. Therefore, the slack between the threads cannot be long. At the same time, a near-zero slack would cause the trailing thread to stall due to unavailable branch outcomes and load values from the leading thread. To support a short slack, SRTR's leading thread provides the trailing thread with branch predictions. In contrast, SRT's leading thread provides branch outcomes. Accordingly, SRTR counts slack in terms of fetched (speculative) instructions, while SRT counts slack in terms of committed instructions.

SRT uses committed values for branch outcomes, load addresses and values, and store addresses and values. Consequently, the StB, LVQ, and BOQ are simple queues that are not affected by mispredictions. Because SRTR compares speculative values of the leading and trailing threads, SRTR needs to handle the effects of mispredictions on these structures.

SRTR uses the SoR that includes the register file. Like SRT, SRTR uses an out-of-order, SMT pipeline [16]. The pipeline places instructions from all threads in a single *issue queue*. Instructions wait in this queue until source operands become available, enabling out-of-order issue. Apart from the issue queue, each thread's instructions are also held in the thread's private *active list (AL)*. When an instruction is issued and removed from the issue queue, the instruction stays in its AL. Instructions commit from the AL in program order, enabling precise interrupts.We illustrate SRTR's SoR and SRTR's additions to SMT in Figure 1.

3.1 Synchronizing leading and trailing threads

For every branch prediction, the leading thread places the predicted PC value in the *prediction queue (predQ)*. This queue is similar to the BOQ except that it holds predictions instead of outcomes. The predQ is emptied in queue-order by the trailing thread. Using the predQ, the two threads fetch essentially the same instructions without diverging.

Because the ALs hold the instructions in predicted program order and because the two threads communicate via the predQ, corresponding leading and trailing instructions occupy the same positions in their respective ALs. Thus, they can be easily located for checking. Note that corresponding leading and trailing instructions may enter their ALs at different times, and become ready to commit at different times. However, we use the fact that the instructions occupy the same position in their ALs to keep the implementation simple.

Due to the slack, the leading and trailing threads resolve

their branches at different times. Upon detecting a misprediction in the leading thread, the leading thread cleans up the predQ, preventing the trailing thread from using mispredicted entries placed earlier in the predQ. There are two possibilities for the timing of events related to a misprediction: (1) The leading branch resolves after the trailing thread has already used the corresponding predQ entry, or (2) the leading branch resolves before the entry is used by the trailing thread.

The first possibility implies that the trailing AL position which mirrors the leading branch's AL position is valid and contains the trailing branch. There are mispredicted trailing instructions in the trailing AL. The leading thread then squashes the mispredicted instructions in the trailing AL, and the existing predQ entries which contain fetch PCs from the incorrect path. Because the leading and trailing ALs are identical, the leading branch can use its own AL pointer to squash the trailing AL. The second possibility implies that the trailing AL position which mirrors the leading branch's AL position is beyond the tail of the trailing AL. In this case, the leading branch squashes all predQ entries later than its own predQ entry, and places the branch outcome in the predQ to be used by the trailing thread later. To prevent faults from causing incorrect squashing, AL pointers are protected by ECC.

Although the leading thread rolls back the predQ and the ALs of both threads upon a misprediction, the leading branch's outcome is still checked against the trailing branch's outcome. The rollback is an optimistic action to reduce the number of mispredicted trailing instructions, assuming that the leading thread is fault-free. If the leading thread's misprediction is incorrectly flagged due to a fault, the trailing branch's check triggers a rollback. We discuss the details of checking in Section 3.3.

3.2 Modifying LVQ

SRT uses a strict queue-ordering for the LVQ, i.e., the leading thread inserts committed load values and addresses at the tail, and the trailing thread empties the load values and addresses from the head of the queue. SRTR modifies SRT's LVQ to operate on speculative cached loads, and therefore, cannot maintain the strict queue order of SRT. To keep the LVQ as compact as possible, we use a table, called the *shadow active list (SAL)*, to hold pointers to LVQ entries (Figure 1(b)). The SAL mirrors the AL in length but is much narrower than the LVQ, and instructions can use the AL pointer to access their information in the SAL. The SAL is also helpful in checking register values as explained in Section 3.3.

A leading load allocates an LVQ entry when the load enters the AL, and places a pointer to the entry in the SAL. Because loads enter the AL in (speculative) program order, LVQ entries are allocated in the same order, simplifying misprediction handling, as explained at the end of the section. Upon issue, the leading load obtains its LVQ pointer from the SAL and places its load value and address in the LVQ. The trailing load also obtains the LVQ pointer from the SAL, and the trailing load's address and the leading load address stored in the LVQ are compared, as done in SRT. On a match, the trailing load obtains the leading load value from the LVQ. A mismatch of the addresses flags a rollback (as explained in Section 3.3) with three possibilities: (1) the address value produced by a previous instruction is faulty and the faulty instruction will initiate a rollback upon being checked; (2) the address computation of the load is faulty and the load instruction will cause rollback to be initiated; (3) the previous instruction was checked and committed and the address register has been corrupted since. Because the register file is inside the SoR, SRTR flags a fault but cannot recover in the third case without protecting the register file with ECC.

Even though leading instructions are fetched and placed ahead of the corresponding trailing instructions in the issue queue, it is possible that a trailing load is issued before the leading load. A possible solution is to place the premature trailing load's address in the empty LVQ entry. When the leading load arrives at the LVQ, the addresses are compared and the pending trailing load is satisfied. This solution naturally extends to the case where a trailing load issues after the leading load, but finds the LVQ entry empty due to the leading load missing in the cache. Note that the LVQ is ECC-protected and so values stored in it are not vulnerable to faults.

An LVQ entry is relinquished in queue order after the trailing instruction reads the entry. Upon a leading branch misprediction, the SAL is rolled back in parallel with the clean-up of the AL. To facilitate the rollback of the LVQ, branches place the LVQ tail pointer in the SAL at the time they enter the AL. Because the LVQ is in (speculative) program order, the LVQ tail pointer points to the LVQ entry to which the LVQ needs to be rolled back, if the branch mispredicts. A mispredicted branch's AL pointer locates the LVQ tail pointer in the SAL, and the LVQ is rolled back to the pointer. Like the predQ's rollback, the LVQ's rollback is also an optimistic action and the leading branch is checked to confirm the misprediction.

3.3 Checking leading and trailing instructions

SRTR checks the leading and trailing instructions as soon as the trailing instruction completes. Register values often have been written back to the physical register file by the time the check is performed. To address this issue, SRTR uses a separate structure, the *register value queue (RVQ)* to store register values for checking, avoiding bandwidth pressure on the register file (Figure 1(b)). In this section, we assume that all the instructions including branches, but excluding loads and stores use the RVQ to be checked. We assume that the RVQ can provide the required bandwidth. We address the bandwidth pressure on the RVQ in Section 4.

Because the trailing instructions need to locate the leading counterpart's value in the RVQ, the leading instruction allocates an RVQ entry at the time of entering the AL, and places a pointer to the entry in the SAL for the trailing instruction. After the leading instruction writes its result back, it enters the *fault-check* stage, as shown in Figure 2. In the fault-check stage, the leading instruction puts its value (for branches, the next PC, the prediction and the outcome are all part of the value) in the RVQ using the pointer from the SAL. The

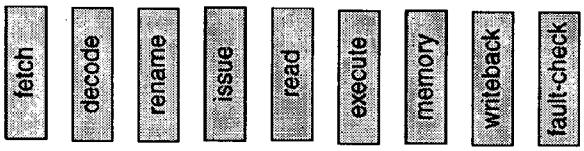


FIGURE 2: SRTR pipeline with fault-check stage.

instruction then waits in the AL to commit or squash due to faults or mispredictions. Because the fault-check stage is *after* writeback, the stage does not affect branch misprediction penalty, or the number of bypass paths.

The trailing instructions also use the SAL to obtain their RVQ pointers and find their leading counterparts' values. While it is likely that the leading instruction reaches the fault-check stage before the trailing instruction, out-of-order pipelines may reverse the order. To handle such reversals, if the trailing instruction finds the RVQ entry of its leading counterpart to be empty, it places its own value. When the leading instruction reaches the fault-check stage, it finds the value and the check is performed. A full/empty bit in the RVQ is used to indicate whether or not the RVQ entry contains leading or trailing values. The full/empty bits need to be ECC-protected to prevent corrupted full/empty bits from causing leading and trailing instruction pairs to wait forever. An RVQ entry is relinquished in queue order after the checking is done. Branch misprediction clean-up of the RVQ uses the SAL in the same way as the clean-up of the LVQ.

SRTR places completed leading store values and addresses in the StB. When a trailing store completes, its value and address are compared to the leading store's value and address in the StB, similar to the checking of load addresses in the LVQ.

Register values, store addresses and values, and load addresses are checked in the RVQ, StB, and LVQ, respectively. If a check succeeds, the *commit vector (CV)* entries corresponding to the leading and trailing instructions are set to the *checked-ok* state. Indexed by AL pointers, the CVs are parallel to the ALs (Figure 1(b)). As instructions are placed in the AL, their CV entries are set to the *not-checked-yet* state. A leading instruction commits only if its CV entry and its trailing counterpart's CV entry are in the checked-ok state. If the state is not-checked-yet, the instruction is stalled at commit until it is checked. Due to the slack, leading instructions may commit ahead of their trailing counterparts.

If a check fails, the CV entries of the leading and trailing instructions are set to the *failed-check* state. When a failed-check entry reaches the head of the leading AL, all later instructions are squashed. The leading thread waits until the trailing thread's corresponding entry reaches the head of the trailing AL before restarting both threads at the offending instruction. Because there is a time gap between the setting and the reading of the CV and between the committing of leading and trailing counterparts, the CV is protected by ECC to prevent faults from corrupting it in the time gap.

There are faults from which SRTR cannot recover: After a register value is written back and the instruction producing the value has committed, if a fault corrupts the register, then the fact that leading and trailing instructions use different physical registers will allow us to detect the fault on the next use of the register value. However, SRTR cannot recover from this fault. To avoid this loss of recovery, one solution is to provide ECC on the register file.

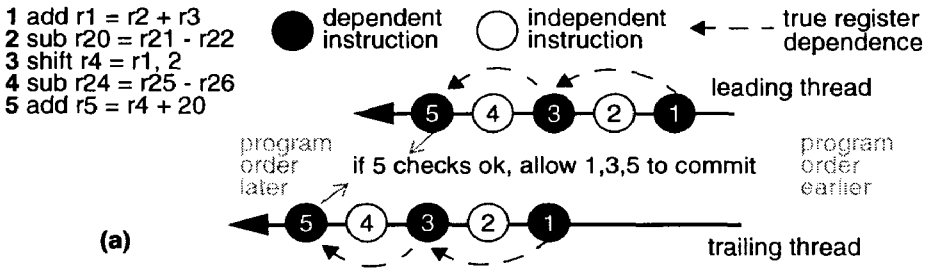
4 Reducing bandwidth demand on RVQ

The RVQ needs to allow as many writes or reads of register values per cycle as the number of leading and trailing non-memory instructions completing in one cycle. Because the RVQ has as many entries as in-flight leading instructions (around 120 64-bit values, totaling to 1KB), providing multiple ports to support high bandwidth (e.g., four 64-bit values per cycle) may be hard.

We propose *dependence-based checking elision (DBCE)* to reduce the number of instructions accessing the RVQ. To keep the implementation simple, we use only simple dependence chains such that each instruction in a chain has at most one parent and one child (instead of maintaining the full dependence graph). By reasoning that faults propagate through dependent instructions, DBCE exploits register (true) dependence chains so that *only* the last instruction in a chain uses the RVQ, and has the leading and trailing values checked. We show an example of a five-instruction sequence in Figure 3(a). The chain's earlier instructions in *both* threads completely elide the RVQ. If the last instruction check succeeds, it signals the previous instructions in the chain that they may commit; if the check fails, all the instructions in the chain are marked as having failed and the earliest instruction in the chain triggers a rollback. A key feature of DBCE is that both leading and trailing instructions redundantly go through the same dependence chain formation and checking-elision decisions, allowing DBCE to check its own functionality for faults.

If the last instruction of a chain is further in the instruction stream and completes much later than the other instructions, the chain's earlier instructions will stall at commit. To avoid this situation, DBCE forms short dependence chains (e.g., 3-4 instructions) by exploiting the abundant register dependencies in near-by instructions. If DBCE's chains are m instructions long, DBCE checks only one out of m instructions, reducing the bandwidth by a factor of m. Because of short slack and short chains, the trailing chain's last instruction completes just a few cycles behind the leading chain's earlier instructions. Consequently, checking of the last instructions is usually done between the time the earliest leading instruction completes and the time it reaches the commit point.

Because leading loads deposit their values in the LVQ for the trailing loads, there is no notion of eliding of checking for load values, and hence loads are not included in the chains. Stores are checked in the StB and do not use the RVQ. Therefore, stores are not included in the chains as well. Because branches do not produce register values, branches cannot be in the middle of a chain. A branch may be at the end of a chain and in that case it itself cannot elide checking but it helps elide checking for the instructions preceding it in its chain. If chains are allowed to cross branches, mispredictions will require



(b)

phy reg	AL ptr	chain tag	flag
101	1	1	1
120	2	2	0
104	3	1	1
124	4	4	0
105	5	1	0

FIGURE 3: (a) DBCE concept. (b) DCQ entries showing only leading instructions.

clean-up of chains whose later instructions have been squashed due to an intervening branch misprediction. To avoid such clean-ups, DBCE disallows chains from crossing branches.

Care must be taken in forming chains with instructions that mask a subset of bits (e.g., r2 := r1 & 0xff00), or that compare two values (e.g., r1 := (r2 < r3)). If a masking (or comparing) instruction is in a chain, then the instruction may mask a fault in its source operands. Such masking violates the key assumption behind DBCE, that faults are propagated by dependences, and a later instruction included in the masking instruction's chain cannot detect the masked fault. For instance, in a chain containing r1 := r2 + r3, r2 := r1 & 0xff00, and r4 := r2 + r3, the check done on the value of r4 does not cover all the bits of r1. If the first instruction produces a faulty value for r1, and r1 is used by instructions other than the one shown above, recovery will not be possible. This issue can be resolved by disallowing masking instructions in the middle of a chain. A masking instruction may, however, start a chain because the instruction's source operands will be checked in previous chains, without allowing any faults to be masked. If an instruction masks some bits for specific inputs (e.g., multiply masks all the bits if one of its source operands is zero), a pessimistic approach would be to disallow the instruction in the middle of a chain. Since the likelihood of such input-specific masking is low, it may be acceptable to ignore the masking and treat the instruction as a non-masking instruction.

Exploiting dependence chains consists of (1) forming dependence chains in the leading thread and the corresponding chains in the trailing thread, and tagging each chain with a unique tag, (2) identifying the instructions in the leading and trailing threads requiring the check, (3) preventing the rest of the instructions (leading and trailing) in the chains from accessing the RVQ and from checking, and (4) notifying the non-checking instructions in the chains after the check is performed.

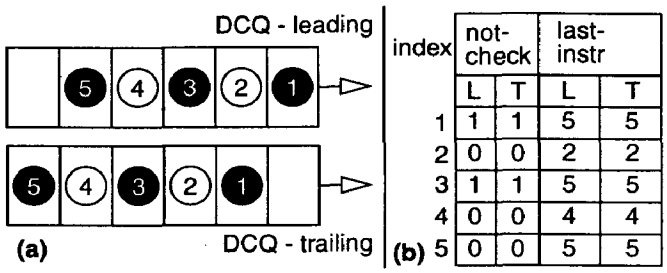
4.1 Forming dependence chains

To form dependence chains, DBCE uses the dependence chain queue (DCQ), which holds information about renamed instructions that were fetched in the last few cycles (e.g., 1-2 cycles). The instructions are kept in the instruction fetch order. Each DCQ entry holds the destination physical register and the AL pointer of an instruction, a tag which identifies the chain to which the instruction belongs, and a flag to indicate if the instruction already has a dependent instruction in its dependence chain. The AL pointer of the first instruction in a chain is used as the chain's tag. Figure 3(b) shows the DCQ entries for only the leading instructions in the example of Figure 3(a), assuming that logical register x maps to physical register 100 + x.

Upon entering the DCQ, an instruction associatively searches the DCQ using its source physical register numbers, matching them against destination physical register numbers of instructions in the DCQ. If there is no match on any source register, or if all the matching instructions already have their flags set indicating that those instructions already have children, there is no live chain to which the current instruction can belong; then the instruction uses its own AL pointer as its tag, and clears its own flag to start a new chain. Branches cannot start a chain, and are removed from the DCQ if they cannot join a live chain. If there is a matching entry with a clear flag, the current instruction adds itself to the matching entry's chain by setting the matching entry's flag and obtaining the matching entry's AL pointer and tag. It clears its flag to allow additional instructions to join the chain. If two sources of an instruction match entries with clear flags, the current instruction adds itself to the chain corresponding to the first source.

Leading and trailing instructions form chains independently. Because there are no dependencies between the two threads, the DCQ can hold the two threads simultaneously. However, because fetching of leading and trailing instructions is interleaved, care must be taken to ensure that the DCQ will form identical dependence chains in the leading and trailing threads. The chains formed may be different if fetch brings a number of leading instructions, switches and brings a smaller number of trailing instructions before switching back to the leading thread. The larger number of leading instructions may cause longer chains to be formed than the fewer trailing instructions.

A simple solution is to have the leading and trailing threads each occupy half the DCQ, as shown in Figure 4(a). Every cycle either leading or trailing thread instructions reach the DCQ. The DCQ evicts the oldest entries of the same thread to make room for the new instructions. Upon evicting the oldest entries, the DCQ terminates the chains originating at the entries, ensuring that the chains stay short and span at most as many cycles as the DCQ depth. The instructions in the terminated chains are recorded in the *check table* for later use. The AL pointer of each of the oldest entries is used to search the DCQ tags, and the matching entries are all the instructions in the chain originating at the oldest entry. Although the number



index	not-check		last-instr	
	L	T	L	T
1	1	1	5	5
2	0	0	2	2
3	1	1	5	5
4	0	0	4	4
5	0	0	5	5

(b)

FIGURE 4: (a) DCQ. (b) Check Table.

of oldest entries may be as large as the fetch width requiring as many parallel searches of the DCQ, the DCQ's small size keeps this parallel search manageable (e.g., 8-way search of 16 entries).

The matching DCQ entry that has a clear flag is the last instruction in the oldest entry's chain. The chain's non-last instructions use their AL pointers to index into the check table. In the table, the non-last instructions set the *not-check* bits to indicate that they elide checking, and they update their *last-instruction* fields with the last instruction's AL pointer. The chain's last instruction keeps its not-check bit clear and updates its last-instruction field with its own AL pointer. Figure 4(b) shows the fields for the example in Figure 3(a), using "L" to denote leading instructions and "T" to denote trailing instructions. Thus, the check table records the chain, identifying the chain's last instruction as needing to check and the rest of the instructions as eliding checking. The check table also associates the eliding instructions with the last instruction whose check covers them.

Operating on renamed instructions guarantees that matching source registers with previous destinations without checking for multiple matches in the DCQ is correct. Also, the DCQ implements a subset of the functionality of renaming and bypass. While it may be possible to use the existing renaming and bypass logic, we describe the DCQ separately for clarity, avoiding implementation details of rename and bypass.

4.2 Using dependence chains

When a leading or trailing instruction reaches writeback, the instruction uses its AL pointer to index into the check table and obtain the AL pointer of the last instruction in its chain. Then the leading and trailing not-check bits in the check table entry are compared. If the bits do not match (indicating a mismatch between the leading and trailing chains caused by a fault), the CV entries of the last instructions in both chains are set to failed-check. Otherwise, if the not-check bit is set, the instruction elides checking and waits in the AL to commit, holding its last instruction AL pointer. If the not-check bit is clear, the leading instruction places its value in the RVQ for later checking. Note that the AL pointers carried by the instructions are ECC-protected. This allows the leading and trailing instructions to access the check table independently without checking against each other's AL pointers.

Later, the trailing instruction indexes into the check table and the not-check bits are compared as above. If both the leading and trailing not-check bits are clear, the leading instruction's value is obtained from the RVQ. The leading and trailing instructions' values are then compared. If the check succeeds, the trailing AL pointer is used to mark the CV entries of both leading and trailing instructions as checked-ok. If the check fails, the CV entries are marked as failed-check. In a chain, only the last-instruction's CV entries are marked, and the rest of the instructions' CV entries remain in the not-checked-yet state. If the trailing instructions reach the RVQ first, then the role of the leading and trailing instructions reverse in the above.

When a leading instruction reaches the head of the AL, its last-instruction's AL pointer is compared to that of its trailing counterpart. On a match, the leading instruction uses its last-instruction's AL pointer to probe the leading and trailing CV entries. Depending on the CV entries, the instruction waits (if the CV entries are not-checked-yet), commits (if the CV entries are checked-ok), or squashes (if the CV entries are failed-check); squashing also occurs on a mismatch. When a trailing instruction reaches the head of its AL, if the leading thread has already committed, the CV entry of the trailing thread is guaranteed to be checked-ok and the trailing instruction will commit.

Upon instruction commit, the check table entry corresponding to the committing instruction's AL pointer are cleared. On an instruction squash, the check table entries corresponding to the squashed AL pointers are invalidated. Because the DCQ holds later instructions down the stream from the squashing instruction, all the DCQ entries from the squashing thread are discarded on a squash.

DBCE needs to guarantee that a dependence chain is fully formed before any of the instructions in the chain enter the writeback stage. Otherwise, when earlier instructions need to know the identity of the last instruction to update the CV and AL, the last instruction will not yet have been identified. We avoid this situation by ensuring that the number of cycles the DCQ spans (1-2 cycles) is smaller than the number of pipeline stages between issue and writeback (usually more than two).

If an interrupt/exception occurs in the middle of a chain, the exception delivery (which is the same as in SRT, as described at the end of Section 2) places the exception at the same execution point in both threads. The excepting instruction and all the previous instructions in the chain wait in the active list for the last instruction in the chain to complete and be checked. Even though the last instruction may be after the excepting instruction in program order, it is acceptable to allow the last instruction to complete but not commit. Only after the checking occurs, is the excepting instruction (and all the previous instructions in the chain) deemed ready to commit. Under this condition, when the excepting instruction reaches the commit point in both threads, an exception is flagged. The rest of the instructions in both threads are now squashed. After the interrupt is handled, execution restarts at the same point for both threads, and new chains are formed afresh. Thus, instructions are committed one by one, exactly as in conventional SMT.

Table 1: Hardware parameters for base system.

Component	Description
Processor	8-way out-of-order issue,128-entry issue queue
Branch prediction	hybrid 8K-entry bimodal, 8K-entry gshare, 8K 2-bit selector 16-entry RAS, 4-way1K BTB (10-cycle misprediction penalty)
L1 I- and D-cache	64KB, 32-byte blocks, 4-way, 2-cycle hit, lock-up free
L2 unified cache	1 Mbyte,64-byte blocks, 4-way, 12-cycle hit, pipelined
Main memory	Infinite capacity, 100 cycle latency

To allow several accesses every cycle, the DCQ has to provide high bandwidth. Because the DCQ holds a small number of instructions, it can be implemented to support high bandwidth (e.g., 8-16 instructions each requiring one 8-bit destination register number, three 8-bit AL pointers, and a few bits for a flag, totaling to about 80 bytes). Building a high-bandwidth RVQ, which is kilobytes in size, is harder, for the same reason that rename tables are widely multiported but D-caches are only dual-ported. The check table is also a high-bandwidth structure because every leading and trailing instruction accesses it. Because the check table holds only two AL pointers and two flags per entry, multi-porting it is easier than multi-porting the much-larger RVQ.

5 Experimental results

We modify the Simplescalar out-of-order simulator [2] to model SMT and SRT. Table 1 shows the base system configuration parameters used throughout the experiments. The front-end of our base pipeline is long to account for SMT and deep pipelines corresponding to high clock speeds. Like SRT, we approximate a high-bandwidth trace cache by fetching past three branches, and at most eight instructions, per cycle [10]. Table 2 presents the SPEC95 benchmarks and their inputs used in this study. In the interest of space, we show results for a subset of the SPEC95 applications, which are representative of our results over the entire SPEC95 suite. We run the benchmarks to completion, or stop at 1 billion instructions in the interest of simulation time.

We present results in the absence of faults in order to study the performance cost of SRTR over SRT. In the presence of faults, SRTR can recover but SRT will stop as soon as it detects a fault. Therefore, a comparison of performance is not possible when faults occur. In addition, faults are expected to be rare enough that the overall performance will be determined by fault-free behavior.

Because the basic scheme used in SRTR for detection is different from that used in SRT, we start by comparing the SRTR detection scheme (without recovery) against SRT. We refer to the detection scheme implemented in SRTR as *SRTRd*. SRTRd uses a short slack and it communicates branch predictions between the leading and trailing threads, while SRT uses a long slack and communicates branch outcomes. In the same experiment, we also show the performance impact of near-zero slack. Then, we report the average time gap between complete to commit, and average memory latencies in the base SMT. These parameters determine the constraints on SRTR's slack which needs to be shorter than the average complete to commit times to avoid leading thread stalls at commit, but long enough to avoid trailing thread stalls due to empty LVQ. We then compare SRTR (providing recovery) using a high-bandwidth RVQ, to SRTRd and SRT. This comparison gives the performance cost of recovery over detection. We show the impact of the RVQ size on SRTR's performance. Finally, we show the bandwidth reduction achieved by DBCE while maintaining the same performance as the high-bandwidth RVQ.

Table 2: Benchmarks and inputs.

Benchmark	Input	#instrs x 10^6	single thread IPC
go	train	600	1.17
lisp	test	1000	1.63
gcc	test	1000	1.28
perl	jumble	1000	1.90
ijpeg	vigo	1000	2.58
vortex	train	1000	2.12
m88ksim	test	500	2.89
compress	train	40	2.16
swim	test	780	2.53
applu	train	680	2.93
fpppp	train	510	0.59
su2cor	test	1000	2.18
hydro2d	test	1000	1.94
tomcatv	test	1000	2.69

5.1 SRT versus SRTRd

In Figure 5, we compare SRTRd and SRT. We show the performance normalized to the base SMT executing only the standard program. We use a slack of 256 and a 256-entry BOQ, 256-entry LVQ and 256-entry StB for SRT, exceeding the sizes for the best performance reported by SRT [10]. For SRTRd, we use predQ/LVQ/StB sizes of 128/128/128 for a slack of 128, 80/96/80 for a slack of 64, 48/96/48 for a slack of 32, and 18/96/18 for a slack of 2. The purpose of this experiment is to compare using a short slack and communicating branch predictions between the leading and trailing threads, against using a long slack and communicating branch outcomes. We do not want the queues to fill up and interfere with this comparison. Therefore, we keep the sizes of SRTRd's queues appropriately large for each slack value. It is important to note that SRTR needs a short slack to avoid leading instructions stalling at commit while waiting for trailing instructions to complete and be checked. This effect does not exist in SRTRd, which performs well with higher values of slack.

From Figure 5, the performance of SRT is between 2% to 45% worse than the base SMT. These numbers are in line with SRT [10]. On average, SRT is 21% worse than SMT for the integer programs (*go* through *compress*), and 27% worse than SMT for the floating point (FP) programs (*swim* through *tomcatv*). In general, programs which exhibit enough ILP to saturate the processor resources in the base SMT incur higher

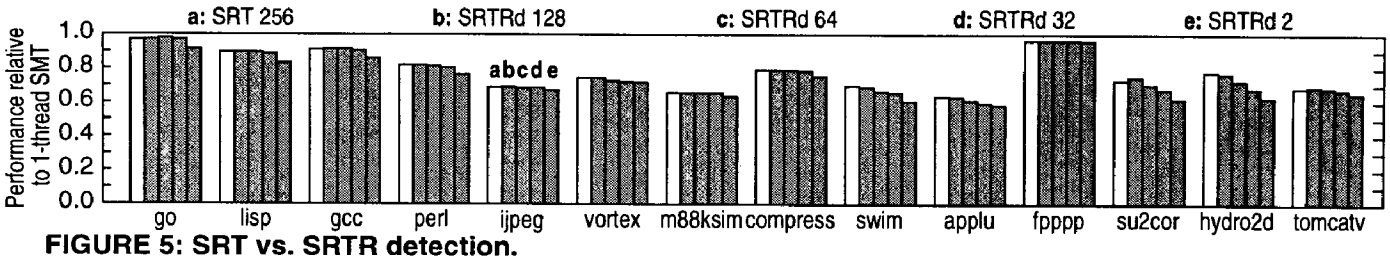


FIGURE 5: SRT vs. SRTR detection.

performance loss due to replication in SRT (and SRTRd).

While a short slack of 64 or 32 performs close to a slack of 256, a near-zero slack of 2 incurs greater performance loss for many programs. For the integer programs, SRTRd on average performs as well as SRT for slacks of 128 and 64, and within 1% of SRT for a slack of 32, showing that communicating branch predictions works as well as outcomes. For a slack of 2, STRTd performs about 5%, on average, worse than SRT, with *lisp* incurring a 10% performance loss. This loss is mainly due to unavailable load values in the LVQ (explained in Section 5.2). For the FP programs, SRTRd performs as well as SRT for a slack of 128, and within 3% and 5% of SRT for a slack of 64 and 32, respectively. For a slack of 2, SRTRd performs about 8%, on average, worse than SRT, with *hydro2d* incurring a 26% performance loss.

In a few cases such as *go*, *tomcatv*, and *su2cor*, SRTRd performs slightly better than SRT due to the interaction between branch mispredictions and slack. Branch mispredictions cause the actual slack to reduce temporarily. In SRT, this reduction results in the trailing thread stalling for branch outcomes, whereas SRTRd is not affected by this reduction because SRTRd's trailing thread uses branch predictions which are available earlier than branch outcomes. Long delays in branch resolution in *tomcatv* and *su2cor*, and *go*'s frequent mispredictions make these programs vulnerable to this effect.

5.2 Constraints on SRTR's slack

While SRTRd performs well with a large slack, recovery will require a shorter slack as discussed earlier. In this section, we explain why a short slack suffices, and how short the slack may be and still not impact performance. For SRT, SRTRd, and SRTR (providing recovery), the slack needs to be long enough to hide the leading thread's average memory latency from the trailing thread. However, SRTR's slack needs to be short enough so that trailing instructions can complete and be checked before the leading counterparts reach the commit point. Hence, SRTR's slack needs to be longer than the memory latency but shorter than the complete to commit time.

Table 3: Slack constraints.

Benchmark	Ave. memory latency	#Ave. complete to commit time
go	2.02	15.5
lisp	2.0	22.8
gcc	2.15	20.5
perl	2.22	27.3
ijpeg	2.15	27.4
vortex	2.15	39.4
m88ksim	2.01	25.4
compress	2.89	26.5
swim	3.36	39.5
applu	3.64	34.3
fpppp	2.0	20.6
su2cor	3.83	40.1
hydro2d	5.80	45.4
tomcatv	2.01	31.3

In Table 3, we tabulate the average number of cycles between complete and commit and the average memory latency for the base SMT. We compute the average memory latency as L1 hit time + L1 miss rate * L1 miss penalty + L2 miss rate * L2 miss penalty. This latency is the impact of the leading thread's load latency on the trailing thread assuming the worst case where the latency is completely exposed in the trailing thread. We see that due to good cache performance, the average memory latency is close to the hit time suggesting that the slack primarily needs to hide on-chip cache hit latencies. In general, the FP programs (*swim* through *tomcatv*) have a higher memory latency explaining their worse performance with shorter slacks. For instance, *hydro2d* has a long average memory latency, and performs poorly with a slack of 2.

For all the programs, the gap between the average complete to commit time and the average memory latency is large enough to allow a slack longer than the average memory latency but shorter than the average complete to commit time. Even for memory-intensive applications which may have higher miss rates than our benchmarks, the gap is likely to be large enough to allow a reasonable slack. Note that slack is counted in number of instructions by which the leading thread is ahead, and the numbers in Table 3 are numbers of cycles. Because fetch can obtain up to 8 instructions per cycle, a slack of 32 is equivalent to 4 cycles.

5.3 SRTR recovery

The average complete to commit times in Table 3 suggest a range for appropriate slack values. To select an acceptable value for the slack, it is important to note that the complete to commit time of individual instructions vary quite widely. For instance, *lisp*, *compress* and *tomcatv* have 40%, 50%, and 40%, respectively, instructions whose complete to commit times are fewer than 10 cycles. Therefore, a long slack may cause many leading instructions to stall at commit waiting for their trailing counterparts to complete and be checked. It is thus important to select a slack value which accommodates the majority of the instructions.

In Figure 6, we compare SRT using a slack of 256 to SRTR

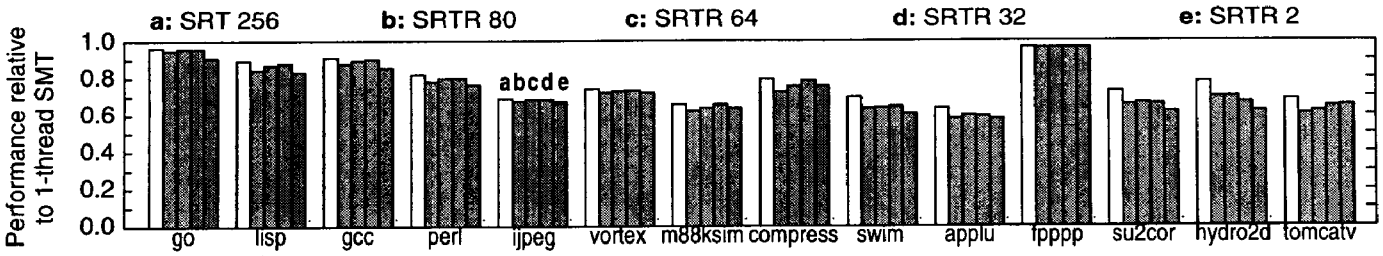


FIGURE 6: SRTR recovery.

(providing recovery) varying the slack for SRTR as 80, 64, 32, and 2. Because SRTRd using a slack of 128 performs as well as SRT, we do not show SRTRd in this graph. To isolate the effect of the slack, we use a bandwidth-unlimited (i.e., 8 ports), 128-entry RVQ (we vary the RVQ size later). We use a 256-entry BOQ, 256-entry LVQ and 256-entry StB for SRT. For SRTR, we use predQ/LVQ/StB sizes of 128/128/128 for a slack of 80, 80/96/80 for a slack of 64, 48/96/48 for a slack of 32, and 18/96/18 for a slack of 2. As in Section 5.1, we show performance normalized to the base SMT executing only the standard program.

It can be seen that SRTR's average performance peaks at a slack of 32. For the integer programs (*go* through *compress*), SRTR using a slack of 64 and 32 on average performs 3% and 1% worse than SRT. For the FP programs (*swim* through *tomcatv*), SRTR on average performs 7% worse than SRT for both a slack of 64 and 32. As expected, decreasing the slack to 2 causes performance degradation. Increasing the slack to 80 also causes performance degradation. SRTR using a slack of 80 on average performs 5% and 9% worse than SRT for the integer and FP programs, respectively. A slack of 80 makes the leading thread stall at commit, putting pressure on the instruction window. Thus, using a slack of 32 seems to be the best choice for these benchmarks.

5.4 RVQ size

In this experiment, we measure the impact of varying the RVQ size on the performance of SRTR. RVQ entries are allocated as leading instructions enter the AL and freed in queue-order as the trailing counterparts obtain the RVQ values. Hence, the RVQ size depends on the issue queue size and the slack. In Figure 7, we compare SRT using a slack of 256 to SRTR using a slack of 32 (which was identified as the best value in the last section) and predQ/LVQ/StB sizes of 48/96/48, but varying the RVQ size as 128, 96, 80, and 64 entries. As before, we show performance normalized to the base SMT executing only the standard program.

It can be seen that an RVQ size of 80 entries works as well as 128 entries for all the programs. With 64 entries, while most programs experience no degradation, a few programs like *gcc*, *compress* and *su2cor* incur a small performance loss while *ijpeg*, *applu*, *hydr2d* and *tomcatv* slow down considerably. For these benchmarks, an RVQ size of 80 entries seems appropriate and achieves the same performance as a 128-entry RVQ.

5.5 DBCE

In this section, we show the effectiveness of DBCE in reducing the bandwidth demand on the RVQ. We measure the impact of RVQ bandwidth on SRTR without and with DBCE. Loads and stores do not use the RVQ and hence the RVQ bandwidth demand comes solely from the ALU/FPU and branch instructions. Both with and without DBCE, SRTR uses a slack of 32, predQ/LVQ/StB sizes of 48/96/48 entries, and an 80-entry RVQ (which was identified as the best size in the last section). We use a DCQ size of 16 (8 for each thread). We varied the DCQ size but did not find much difference mainly because the chains are broken at branches, and branch frequency impacts the chain length more than the DCQ size. Because four RVQ ports are as good as five or more for SRTR without DBCE, we vary the number of RVQ ports as 2, 3, and 4. We use SRT with a slack of 256 as the reference, and show performance normalized to the base SMT executing only the standard program.

We show the results in Figure 8. In our implementation, we assume that faults propagate through all instructions, including those that may mask faults. Because the percentage of masking instructions is usually low [10], this assumption will not affect our results significantly.

In Table 4, we show the number of RVQ accesses elided by DBCE as a percentage of all RVQ accesses made without DBCE. On average, DBCE elides 35.3% of all RVQ accesses in both leading and trailing threads. For most programs, the percentage of elided instructions is high using a DCQ of just 16 entries because the programs have an abundance of register

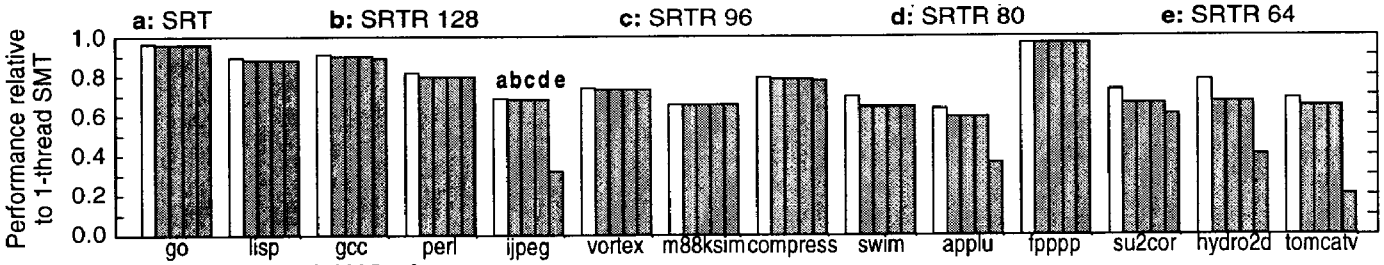


FIGURE 7: Impact of RVQ size.

Table 4: Percent RVQ accesses elided.

Benchmark	Percent elided	Benchmark	Percent elided
go	53.1	swim	43.7
lisp	24.7	applu	50.1
gcc	41.3	fpppp	18.4
perl	33.5	su2cor	40.8
ijpeg	49.4	hydro2d	39.5
vortex	15.7	tomcatv	35.1
m88ksim	38.4	AVERAGE	35.3
compress	38.5		

dependences in nearby instructions. The exceptions are *vortex* and *fpppp*; both programs have a high fraction (52.8% and 53.2%, respectively) of memory instructions. Because loads and stores are not included in the DBCE chains, the programs cannot elide as many instructions as the others.

Let us first analyze SRTR performance without DBCE. From Figure 8, we see that for all the programs, a 4-ported RVQ (third bar) performs as well as an 8-ported RVQ (second bar). As the number of RVQ ports decreases from 3 to 2, most programs incur significant performance loss. For the integer programs (*go* thorough *compress*), performance drops by 2% and 18%, on average, with 3 and 2 RVQ ports, respectively, compared to 4 RVQ ports. For the FP programs (*swim* through *tomcatv*), performance degrades by 1% and 20%, on average, with 3 and 2 RVQ ports, respectively. *vortex* and *fpppp* are the two exceptions that perform as well with 2 RVQ ports as with 4 RVQ ports, because more than half of the instructions are loads and stores, and do not access the RVQ.

On the other hand, SRTR with DBCE incurs little performance loss even with two RVQ ports. Comparing four ports to two ports, performance degrades by 1% and 2% for the integer and FP programs, respectively. Note that in the case of 4 ports where DBCE is not needed, using DBCE does not degrade performance. This point implies that by exploiting complete to commit time, DBCE avoids stalling the early instructions in the chains waiting for the last instruction in the chain to complete. Looking at SRTR using 2 RVQ ports with and without DBCE, DBCE boosts SRTR's performance by 17% and 18% for the integer and FP programs, respectively.

6 Related work

Watchdog processors are the key concept behind many fault tolerance schemes [5]. The AR-SMT processor is the first to use SMT to execute two copies of the same program [12]. AR-SMT also proposes using speculation techniques to allow communication of data values and branch outcomes between the leading and trailing threads to accelerate execution. A later paper applies the concepts from AR-SMT to CMPs [15]. SRT improves on AR-SMT via the two optimizations of slack fetch and checking only stores for an SoR that includes the register file [10]. A recent paper explores design options for fault detection via multithreading [6].

The AR-SMT paper mentions recovery stating that the state of the R-stream (which corresponds to our trailing thread) is the checkpointed state and can be used for recovery. SRTR and AR-SMT are fundamentally different ways of performing recovery, with different costs. SRTR disallows the leading thread from committing until the trailing thread completes and is checked, and uses instruction squash to rollback to a committed state before the fault. AR-SMT allows the leading thread to commit potentially faulty state, and let the trailing thread be checked upon completion of each instruction; upon detecting a fault, AR-SMT uses the trailing thread's committed state (up to but not including the fault) to restore the leading thread's state for recovery. SRTR delays the leading thread from committing and our paper shows the performance impact of this choice. AR-SMT doubles the bandwidth pressure on the data cache by requiring both threads to access the cache, while SRTR (and SRT) uses the LVQ for the trailing thread accesses. Furthermore, AR-SMT requires memory to be doubled (two copies of memory, one for each thread) because committing faulty state of the leading thread will corrupts memory. Doubling the memory size may stress the memory hierarchy and degrade performance. Because faults are not allowed to reach memory in SRTR, there is only one copy of memory in SRTR (and SRT).

DIVA is another fault-tolerant superscalar processor that uses a simple, in-order checker processor to check the execution of the complex out-of-order processor [1]. DIVA can recover from permanent faults and design errors in the aggressive processor but assumes that no transient faults occur in the checker processor itself. Other works on fault tolerance focus on functional units [11, 7, 4, 14].

A recent paper [9] proposes hardware recovery using superscalar hardware without any SMT support. The paper advocates the natural way to achieve recovery by using superscalar's rollback ability. The paper does not use the LVQ, does not address the issues related to cached loads, and claims that there is no need for any slack.

The Compaq NonStop Himalaya [3] and IBM z900 (formerly S/390) [13] employ redundant hardware to achieve fault

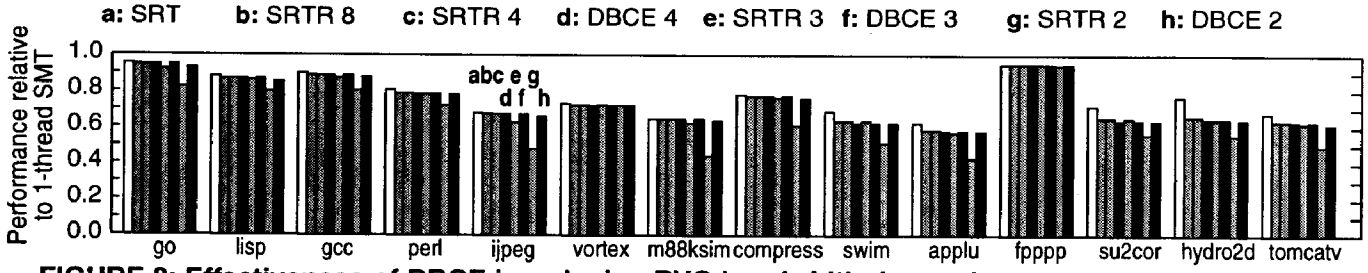


FIGURE 8: Effectiveness of DBCE in reducing RVQ bandwidth demand.

tolerance. The z900 uses the G5 microprocessor which includes replicated, lock-stepped pipelines. The NonStop Himalaya uses off-the-shelf, lock-stepped microprocessors and compares the external pins on every cycle. In both systems, when the lock-stepped components disagree, the components are stopped to prevent propagation of faults. The z900 uses special microcode to restore program state from a hardware checkpoint module. The NonStop Himalaya does not provide hardware support for recovery. SRT has shown that avoiding lock-stepping achieves better performance.

7 Conclusions

We proposed *Simultaneously and Redundantly Threaded processors with Recovery (SRTR)* that enhances SRT to include transient-fault recovery. In SRT, a leading instruction may commit *before* the check for faults occurs, relying on the trailing thread to trigger detection. SRTR, on the other hand, must *not* allow *any* leading instruction to commit before checking occurs, since a faulty instruction cannot be undone once the instruction commits. To avoid leading instructions stalling at commit waiting for their trailing counterparts, SRTR exploits the time between completion and commit of a leading instruction. SRTR checks as soon as the trailing instruction completes, well before the leading instruction reaches commit. To avoid increasing the bandwidth demand on the register file, SRTR uses the *register value queue (RVQ)* to hold register values for checking. To reduce the bandwidth pressure on the RVQ itself, SRTR employs *dependence-based checking elision (DBCE)*. By reasoning that faults propagate through dependent instructions, DBCE exploits register (true) dependence chains so that *only* the last instruction in a chain uses the RVQ, and has the leading and trailing values checked. DBCE redundantly builds chains in both the leading and trailing threads and checks its own functionality for faults.

We evaluated SRTR using the SPEC95 benchmarks. SRTR on average performs within 1% and 7% of SRT for integer and floating-point programs, respectively. We showed that high prediction accuracies and low off-chip miss rates in the underlying SMT enable SRTR detection using a slack of 32 to perform on average within 5% of SRT using a slack of 256. For our benchmarks, the gap between the average complete to commit time and average memory latency is large enough to allow a slack longer than the average memory latency but shorter than the average complete to commit time. DBCE elides about 35% of RVQ accesses. SRTR without DBCE on average incurs 18% performance loss on reducing from four (which is performance-equivalent to an unlimited number) to two RVQ ports. With DBCE, a two-ported RVQ on average performs within 2% of a four-ported RVQ.

Acknowledgements

We thank Shubu Mukherjee and the anonymous reviewers for their comments. The work of I. Pomeranz and K. Cheng was supported in part by NSF Grant No. CCR-0049081. The work of T. N. Vijaykumar was supported in part by NSF Grant No. CCR-9875960.

References

[1] T. M. Austin. DIVA: A reliable substrate for deep-submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report CS TR-1308, University of Wisconsin, Madison, July 1996.

[3] Compaq Computer Corporation. *Data integrity for Compaq Non-Stop Himalaya servers.* http://nonstop.compaq.com, 1999.

[4] J. G. Holm and P. Banerjee. Low cost concurrent error detection in a VLIW architecture using replicated instructions. In *Proceedings of the International Conference on Parallel Processing*, 1992.

[5] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors–A survey. *IEEE Trans. on Computers*, 37(2):160–174, Feb. 1988.

[6] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[7] J. H. Patel and L. Y. Fung. Concurrent error detection on ALU's by recomputing with shifted operands. *IEEE Trans. on Computers*, 31(7):589–595, July 1982.

[8] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1998.

[9] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual IEEE/ACM international symposium o n Microarchitecture*, Dec. 2001.

[10] S. K. Reinhardt and S. S. Mukherjee. Transient-fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[11] D. A. Reynolds and G. Metze. Fault detection capabilities of alternating logic. *IEEE Trans. on Computers*, 27(12):1093–1098, Dec. 1978.

[12] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of Fault-Tolerant Computing Systems*, 1999.

[13] T. J. Slegel, et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[14] G. S. Sohi, M. Franklin, and K. K. Saluja. A study of time-redundant fault tolerance techniques for high-performance, pipelined computers. In *Digest of papers, 19th International Symposium on Fault-Tolerant Computing*, pages 436–443, 1989.

[15] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault-tolerance. In *Proceedings of the Ninth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 257–268. Association for Computing Machinery, Nov. 2000.

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

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